

# Arm<sup>®</sup> A64 Instruction Set

## for A-profile architecture

**arm**

# Arm A64 Instruction Set for A-profile architecture

Copyright © 2010-2023 Arm Limited (or its affiliates). All rights reserved.

## Release Information

For information on the change history and known issues for this release, see the **Release Notes** in the **A64 ISA XML for A-profile architecture (2023-06)**.

## Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

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

Copyright © 2010-2023 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.  
110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349 version 21.0)

## Confidentiality Status

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

## Product Status

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

The information relating to FEAT\_MEC, the 2022 Extensions of the A-profile Architecture (except for FEAT\_GCS, FEAT\_D128, and the Debug and PMU 2022 features), and the rest of the Architecture is at Beta quality. Beta quality means that all major features of the specification are described, but some details might be missing.

The information relating to FEAT\_GCS, FEAT\_D128, and the Debug and PMU 2022 features remains at Alpha quality. Alpha quality means that most major features of the specification are described in this release, but some features and details might be missing.

### **Web Address**

<http://www.arm.com>

### **Progressive Terminology Commitment**

Previous issues of this document included terms that can be offensive. We have replaced these terms. If you find offensive terms in this document, please contact [terms@arm.com](mailto:terms@arm.com).

### **Feedback on this document**

If you have any comments or queries about this document, create a ticket at <https://support.developer.arm.com>.

As part of the ticket, include:

- The title, *Arm® A64 Instruction Set for A-profile architecture*.
- The number, DDI 0596.
- The section name to which your comments refer.
- The page number(s) to which your comments refer.
- The rule identifier(s) to which your comments refer if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.





## A64 -- Base Instructions (alphabetic order)

[ABS](#): Absolute value.

[ADC](#): Add with Carry.

[ADCS](#): Add with Carry, setting flags.

[ADD \(extended register\)](#): Add (extended register).

[ADD \(immediate\)](#): Add (immediate).

[ADD \(shifted register\)](#): Add (shifted register).

[ADDG](#): Add with Tag.

[ADDS \(extended register\)](#): Add (extended register), setting flags.

[ADDS \(immediate\)](#): Add (immediate), setting flags.

[ADDS \(shifted register\)](#): Add (shifted register), setting flags.

[ADR](#): Form PC-relative address.

[ADRP](#): Form PC-relative address to 4KB page.

[AND \(immediate\)](#): Bitwise AND (immediate).

[AND \(shifted register\)](#): Bitwise AND (shifted register).

[ANDS \(immediate\)](#): Bitwise AND (immediate), setting flags.

[ANDS \(shifted register\)](#): Bitwise AND (shifted register), setting flags.

[ASR \(immediate\)](#): Arithmetic Shift Right (immediate): an alias of SBFM.

[ASR \(register\)](#): Arithmetic Shift Right (register): an alias of ASRV.

[ASRV](#): Arithmetic Shift Right Variable.

[AT](#): Address Translate: an alias of SYS.

[AUTDA, AUTDZA](#): Authenticate Data address, using key A.

[AUTDB, AUTDZB](#): Authenticate Data address, using key B.

[AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA](#): Authenticate Instruction address, using key A.

[AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB](#): Authenticate Instruction address, using key B.

[AXFLAG](#): Convert floating-point condition flags from Arm to external format.

[B](#): Branch.

[B.cond](#): Branch conditionally.

[BC.cond](#): Branch Consistent conditionally.

[BFC](#): Bitfield Clear: an alias of BFM.

[BFI](#): Bitfield Insert: an alias of BFM.

[BFM](#): Bitfield Move.

[BFXIL](#): Bitfield extract and insert at low end: an alias of BFM.

[BIC \(shifted register\)](#): Bitwise Bit Clear (shifted register).

[BICS \(shifted register\)](#): Bitwise Bit Clear (shifted register), setting flags.

[BL](#): Branch with Link.

[BLR](#): Branch with Link to Register.

[BLRAA](#), [BLRAAZ](#), [BLRAB](#), [BLRABZ](#): Branch with Link to Register, with pointer authentication.

[BR](#): Branch to Register.

[BRAA](#), [BRAAZ](#), [BRAB](#), [BRABZ](#): Branch to Register, with pointer authentication.

[BRB](#): Branch Record Buffer: an alias of SYS.

[BRK](#): Breakpoint instruction.

[BTI](#): Branch Target Identification.

[CAS](#), [CASA](#), [CASAL](#), [CASL](#): Compare and Swap word or doubleword in memory.

[CASB](#), [CASAB](#), [CASALB](#), [CASLB](#): Compare and Swap byte in memory.

[CASH](#), [CASAHL](#), [CASALH](#), [CASLH](#): Compare and Swap halfword in memory.

[CASP](#), [CASPA](#), [CASPAL](#), [CASPL](#): Compare and Swap Pair of words or doublewords in memory.

[CBNZ](#): Compare and Branch on Nonzero.

[CBZ](#): Compare and Branch on Zero.

[CCMN \(immediate\)](#): Conditional Compare Negative (immediate).

[CCMN \(register\)](#): Conditional Compare Negative (register).

[CCMP \(immediate\)](#): Conditional Compare (immediate).

[CCMP \(register\)](#): Conditional Compare (register).

[CFINV](#): Invert Carry Flag.

[CFP](#): Control Flow Prediction Restriction by Context: an alias of SYS.

[CHKFEAT](#): Check feature status.

[CINC](#): Conditional Increment: an alias of CSINC.

[CINV](#): Conditional Invert: an alias of CSINV.

[CLRBHB](#): Clear Branch History.

[CLREX](#): Clear Exclusive.

[CLS](#): Count Leading Sign bits.

[CLZ](#): Count Leading Zeros.

[CMN \(extended register\)](#): Compare Negative (extended register): an alias of ADDS (extended register).

[CMN \(immediate\)](#): Compare Negative (immediate): an alias of ADDS (immediate).

[CMN \(shifted register\)](#): Compare Negative (shifted register): an alias of ADDS (shifted register).

[CMP \(extended register\)](#): Compare (extended register): an alias of SUBS (extended register).

[CMP \(immediate\)](#): Compare (immediate): an alias of SUBS (immediate).

[CMP \(shifted register\)](#): Compare (shifted register): an alias of SUBS (shifted register).

[CMPP](#): Compare with Tag: an alias of SUBPS.

[CNEG](#): Conditional Negate: an alias of CSNEG.

[CNT](#): Count bits.

[COSP](#): Clear Other Speculative Prediction Restriction by Context: an alias of SYS.

[CPP](#): Cache Prefetch Prediction Restriction by Context: an alias of SYS.

[CPYFP](#), [CPYFM](#), [CPYFE](#): Memory Copy Forward-only.

[CPYFPN](#), [CPYFMN](#), [CPYFEN](#): Memory Copy Forward-only, reads and writes non-temporal.

[CPYFPRN](#), [CPYFMRN](#), [CPYFERN](#): Memory Copy Forward-only, reads non-temporal.

[CPYFPRT](#), [CPYFMRT](#), [CPYFERT](#): Memory Copy Forward-only, reads unprivileged.

[CPYFPRTN](#), [CPYFMRTN](#), [CPYFERTN](#): Memory Copy Forward-only, reads unprivileged, reads and writes non-temporal.

[CPYFPRTN](#), [CPYFMRTN](#), [CPYFERTN](#): Memory Copy Forward-only, reads unprivileged and non-temporal.

[CPYFPRTWN](#), [CPYFMRTWN](#), [CPYFERTWN](#): Memory Copy Forward-only, reads unprivileged, writes non-temporal.

[CPYFPT](#), [CPYFMT](#), [CPYFET](#): Memory Copy Forward-only, reads and writes unprivileged.

[CPYFPTN](#), [CPYFMTN](#), [CPYFETN](#): Memory Copy Forward-only, reads and writes unprivileged and non-temporal.

[CPYFPTRN](#), [CPYFMTRN](#), [CPYFETRN](#): Memory Copy Forward-only, reads and writes unprivileged, reads non-temporal.

[CPYFPTWN](#), [CPYFMTWN](#), [CPYFETWN](#): Memory Copy Forward-only, reads and writes unprivileged, writes non-temporal.

[CPYFPWN](#), [CPYFMWN](#), [CPYFEWN](#): Memory Copy Forward-only, writes non-temporal.

[CPYFPWT](#), [CPYFMWT](#), [CPYFEWT](#): Memory Copy Forward-only, writes unprivileged.

[CPYFPWTN](#), [CPYFMWTN](#), [CPYFEWTN](#): Memory Copy Forward-only, writes unprivileged, reads and writes non-temporal.

[CPYFPWTRN](#), [CPYFMWTRN](#), [CPYFEWTRN](#): Memory Copy Forward-only, writes unprivileged, reads non-temporal.

[CPYFPWTWN](#), [CPYFMWTWN](#), [CPYFEWTWN](#): Memory Copy Forward-only, writes unprivileged and non-temporal.

[CPYP](#), [CPYM](#), [CPYE](#): Memory Copy.

[CPYPN](#), [CPYMN](#), [CPYEN](#): Memory Copy, reads and writes non-temporal.

[CPYPRN](#), [CPYMRN](#), [CPYERN](#): Memory Copy, reads non-temporal.

[CPYPRT](#), [CPYMRT](#), [CPYERT](#): Memory Copy, reads unprivileged.

[CPYPRTN](#), [CPYMRTN](#), [CPYERTN](#): Memory Copy, reads unprivileged, reads and writes non-temporal.

[CPYPRTN](#), [CPYMRTN](#), [CPYERTN](#): Memory Copy, reads unprivileged and non-temporal.

[CPYPRTWN](#), [CPYMRTWN](#), [CPYERTWN](#): Memory Copy, reads unprivileged, writes non-temporal.

[CPYPT](#), [CPYMT](#), [CPYET](#): Memory Copy, reads and writes unprivileged.

[CPYPTN](#), [CPYMTN](#), [CPYETN](#): Memory Copy, reads and writes unprivileged and non-temporal.

[CPYPTRN](#), [CPYMTRN](#), [CPYETRN](#): Memory Copy, reads and writes unprivileged, reads non-temporal.

[CPYPTWN](#), [CPYMTWN](#), [CPYETWN](#): Memory Copy, reads and writes unprivileged, writes non-temporal.

[CPYPWN](#), [CPYMWN](#), [CPYEWN](#): Memory Copy, writes non-temporal.

[CPYPWT](#), [CPYMWT](#), [CPYEWT](#): Memory Copy, writes unprivileged.

[CPYPWTN](#), [CPYMWTN](#), [CPYEWTN](#): Memory Copy, writes unprivileged, reads and writes non-temporal.

[CPYPWTRN](#), [CPYMWTRN](#), [CPYEWTRN](#): Memory Copy, writes unprivileged, reads non-temporal.

[CPYPWTWN](#), [CPYMWWTWN](#), [CPYEWWTWN](#): Memory Copy, writes unprivileged and non-temporal.

[CRC32B](#), [CRC32H](#), [CRC32W](#), [CRC32X](#): CRC32 checksum.

[CRC32CB](#), [CRC32CH](#), [CRC32CW](#), [CRC32CX](#): CRC32C checksum.

[CSDB](#): Consumption of Speculative Data Barrier.

[CSEL](#): Conditional Select.

[CSET](#): Conditional Set: an alias of CSINC.

[CSETM](#): Conditional Set Mask: an alias of CSINV.

[CSINC](#): Conditional Select Increment.

[CSINV](#): Conditional Select Invert.

[CSNEG](#): Conditional Select Negation.

[CTZ](#): Count Trailing Zeros.

[DC](#): Data Cache operation: an alias of SYS.

[DCPS1](#): Debug Change PE State to EL1.

[DCPS2](#): Debug Change PE State to EL2.

[DCPS3](#): Debug Change PE State to EL3.

[DGH](#): Data Gathering Hint.

[DMB](#): Data Memory Barrier.

[DRPS](#): Debug restore process state.

[DSB](#): Data Synchronization Barrier.

[DVP](#): Data Value Prediction Restriction by Context: an alias of SYS.

[EON \(shifted register\)](#): Bitwise Exclusive-OR NOT (shifted register).

[EOR \(immediate\)](#): Bitwise Exclusive-OR (immediate).

[EOR \(shifted register\)](#): Bitwise Exclusive-OR (shifted register).

[ERET](#): Exception Return.

[ERETAA](#), [ERETAB](#): Exception Return, with pointer authentication.

[ESB](#): Error Synchronization Barrier.

[EXTR](#): Extract register.

[GCSB DSYNC](#): Guarded Control Stack Barrier.

[GCSPOPCX](#): Guarded Control Stack Pop and Compare exception return record: an alias of SYS.

[GCSPOPM](#): Guarded Control Stack Pop: an alias of SYSL.

[GCSPOPX](#): Guarded Control Stack Pop exception return record: an alias of SYS.

[GCSPUSHM](#): Guarded Control Stack Push: an alias of SYS.

[GCSPUSHX](#): Guarded Control Stack Push exception return record: an alias of SYS.

[GCSSS1](#): Guarded Control Stack Switch Stack 1: an alias of SYS.

[GCSSS2](#): Guarded Control Stack Switch Stack 2: an alias of SYSL.

[GCSSTR](#): Guarded Control Stack Store.

[GCSSTTR](#): Guarded Control Stack unprivileged Store.

[GMI](#): Tag Mask Insert.

[HINT](#): Hint instruction.

[HLT](#): Halt instruction.

[HVC](#): Hypervisor Call.

[IC](#): Instruction Cache operation: an alias of SYS.

[IRG](#): Insert Random Tag.

[ISB](#): Instruction Synchronization Barrier.

[LD64B](#): Single-copy Atomic 64-byte Load.

[LDADD](#), [LDADDA](#), [LDADDAL](#), [LDADDL](#): Atomic add on word or doubleword in memory.

[LDADDB](#), [LDADDAB](#), [LDADDALB](#), [LDADDLB](#): Atomic add on byte in memory.

[LDADDH](#), [LDADDAH](#), [LDADDALH](#), [LDADDLH](#): Atomic add on halfword in memory.

[LDAPR](#): Load-Acquire RCpc Register.

[LDAPRB](#): Load-Acquire RCpc Register Byte.

[LDAPRH](#): Load-Acquire RCpc Register Halfword.

[LDAPUR](#): Load-Acquire RCpc Register (unscaled).

[LDAPURB](#): Load-Acquire RCpc Register Byte (unscaled).

[LDAPURH](#): Load-Acquire RCpc Register Halfword (unscaled).

[LDAPURSB](#): Load-Acquire RCpc Register Signed Byte (unscaled).

[LDAPURSH](#): Load-Acquire RCpc Register Signed Halfword (unscaled).

[LDAPURSW](#): Load-Acquire RCpc Register Signed Word (unscaled).

[LDAR](#): Load-Acquire Register.

[LDARB](#): Load-Acquire Register Byte.

[LDARH](#): Load-Acquire Register Halfword.

[LDAXP](#): Load-Acquire Exclusive Pair of Registers.

[LDAXR](#): Load-Acquire Exclusive Register.

[LDAXRB](#): Load-Acquire Exclusive Register Byte.

[LDAXRH](#): Load-Acquire Exclusive Register Halfword.

[LDCLR](#), [LDCLRA](#), [LDCLRAL](#), [LDCLRL](#): Atomic bit clear on word or doubleword in memory.

[LDCLRB](#), [LDCLRAB](#), [LDCLRALB](#), [LDCLRLB](#): Atomic bit clear on byte in memory.

[LDCLRH](#), [LDCLRAH](#), [LDCLRALH](#), [LDCLRLH](#): Atomic bit clear on halfword in memory.

[LDCLRE](#), [LDCLRPA](#), [LDCLRPAL](#), [LDCLRPL](#): Atomic bit clear on quadword in memory.

[LDEOR](#), [LDEORA](#), [LDEORAL](#), [LDEORL](#): Atomic Exclusive-OR on word or doubleword in memory.

[LDEORB](#), [LDEORAB](#), [LDEORALB](#), [LDEORLB](#): Atomic Exclusive-OR on byte in memory.

[LDEORH](#), [LDEORAH](#), [LDEORALH](#), [LDEORLH](#): Atomic Exclusive-OR on halfword in memory.

[LDG](#): Load Allocation Tag.

[LDGM](#): Load Tag Multiple.

[LDIAPP](#): Load-Acquire RCpc ordered Pair of registers.

[LDLAR](#): Load LOAcquire Register.

[LDLARB](#): Load LOAcquire Register Byte.

[LDLARH](#): Load LOAcquire Register Halfword.

[LDNP](#): Load Pair of Registers, with non-temporal hint.

[LDP](#): Load Pair of Registers.

[LDPSW](#): Load Pair of Registers Signed Word.

[LDR \(immediate\)](#): Load Register (immediate).

[LDR \(literal\)](#): Load Register (literal).

[LDR \(register\)](#): Load Register (register).

[LDRAA](#), [LDRAB](#): Load Register, with pointer authentication.

[LDRB \(immediate\)](#): Load Register Byte (immediate).

[LDRB \(register\)](#): Load Register Byte (register).

[LDRH \(immediate\)](#): Load Register Halfword (immediate).

[LDRH \(register\)](#): Load Register Halfword (register).

[LDRSB \(immediate\)](#): Load Register Signed Byte (immediate).

[LDRSB \(register\)](#): Load Register Signed Byte (register).

[LDRSH \(immediate\)](#): Load Register Signed Halfword (immediate).

[LDRSH \(register\)](#): Load Register Signed Halfword (register).

[LDRSW \(immediate\)](#): Load Register Signed Word (immediate).

[LDRSW \(literal\)](#): Load Register Signed Word (literal).

[LDRSW \(register\)](#): Load Register Signed Word (register).

[LDSET](#), [LDSETA](#), [LDSETAL](#), [LDSETL](#): Atomic bit set on word or doubleword in memory.

[LDSETB](#), [LDSETAB](#), [LDSETALB](#), [LDSETLB](#): Atomic bit set on byte in memory.

[LDSETH](#), [LDSETAH](#), [LDSETALH](#), [LDSETLH](#): Atomic bit set on halfword in memory.

[LDSETP](#), [LDSETPA](#), [LDSETPAL](#), [LDSETPL](#): Atomic bit set on quadword in memory.

[LDSMAX](#), [LDSMAXA](#), [LDSMAXAL](#), [LDSMAXL](#): Atomic signed maximum on word or doubleword in memory.

[LDSMAXB](#), [LDSMAXAB](#), [LDSMAXALB](#), [LDSMAXLB](#): Atomic signed maximum on byte in memory.

[LDSMAXH](#), [LDSMAXAH](#), [LDSMAXALH](#), [LDSMAXLH](#): Atomic signed maximum on halfword in memory.

[LDSMIN](#), [LDSMINA](#), [LDSMINAL](#), [LDSMINL](#): Atomic signed minimum on word or doubleword in memory.

[LDSMINB](#), [LDSMINAB](#), [LDSMINALB](#), [LDSMINLB](#): Atomic signed minimum on byte in memory.

[LDSMINH](#), [LDSMINAH](#), [LDSMINALH](#), [LDSMINLH](#): Atomic signed minimum on halfword in memory.

[LDTR](#): Load Register (unprivileged).

[LDTRB](#): Load Register Byte (unprivileged).

[LDTRH](#): Load Register Halfword (unprivileged).

[LDTRSB](#): Load Register Signed Byte (unprivileged).

[LDTRSH](#): Load Register Signed Halfword (unprivileged).

[LDTRSW](#): Load Register Signed Word (unprivileged).

[LDUMAX](#), [LDUMAXA](#), [LDUMAXAL](#), [LDUMAXL](#): Atomic unsigned maximum on word or doubleword in memory.

[LDUMAXB](#), [LDUMAXAB](#), [LDUMAXALB](#), [LDUMAXLB](#): Atomic unsigned maximum on byte in memory.

[LDUMAXH](#), [LDUMAXAH](#), [LDUMAXALH](#), [LDUMAXLH](#): Atomic unsigned maximum on halfword in memory.

[LDUMIN](#), [LDUMINA](#), [LDUMINAL](#), [LDUMINL](#): Atomic unsigned minimum on word or doubleword in memory.

[LDUMINB](#), [LDUMINAB](#), [LDUMINALB](#), [LDUMINLB](#): Atomic unsigned minimum on byte in memory.

[LDUMINH](#), [LDUMINAH](#), [LDUMINALH](#), [LDUMINLH](#): Atomic unsigned minimum on halfword in memory.

[LDUR](#): Load Register (unscaled).

[LDURB](#): Load Register Byte (unscaled).

[LDURH](#): Load Register Halfword (unscaled).

[LDURSB](#): Load Register Signed Byte (unscaled).

[LDURSH](#): Load Register Signed Halfword (unscaled).

[LDURSW](#): Load Register Signed Word (unscaled).

[LDXP](#): Load Exclusive Pair of Registers.

[LDXR](#): Load Exclusive Register.

[LDXRB](#): Load Exclusive Register Byte.

[LDXRH](#): Load Exclusive Register Halfword.

[LSL \(immediate\)](#): Logical Shift Left (immediate): an alias of UBFM.

[LSL \(register\)](#): Logical Shift Left (register): an alias of LSLV.

[LSLV](#): Logical Shift Left Variable.

[LSR \(immediate\)](#): Logical Shift Right (immediate): an alias of UBFM.

[LSR \(register\)](#): Logical Shift Right (register): an alias of LSRV.

[LSRV](#): Logical Shift Right Variable.

[MADD](#): Multiply-Add.

[MNEG](#): Multiply-Negate: an alias of MSUB.

[MOV \(bitmask immediate\)](#): Move (bitmask immediate): an alias of ORR (immediate).

[MOV \(inverted wide immediate\)](#): Move (inverted wide immediate): an alias of MOVN.

[MOV \(register\)](#): Move (register): an alias of ORR (shifted register).

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

[MOV \(wide immediate\)](#): Move (wide immediate): an alias of MOVZ.

[MOVK](#): Move wide with keep.

[MOVN](#): Move wide with NOT.

[MOVZ](#): Move wide with zero.

[MRRS](#): Move System Register to two adjacent general-purpose registers.

[MRS](#): Move System Register to general-purpose register.

[MSR \(immediate\)](#): Move immediate value to Special Register.

[MSR \(register\)](#): Move general-purpose register to System Register.

[MSRR](#): Move two adjacent general-purpose registers to System Register.

[MSUB](#): Multiply-Subtract.

[MUL](#): Multiply: an alias of MADD.

[MVN](#): Bitwise NOT: an alias of ORN (shifted register).

[NEG \(shifted register\)](#): Negate (shifted register): an alias of SUB (shifted register).

[NEGS](#): Negate, setting flags: an alias of SUBS (shifted register).

[NGC](#): Negate with Carry: an alias of SBC.

[NGCS](#): Negate with Carry, setting flags: an alias of SBCS.

[NOP](#): No Operation.

[ORN \(shifted register\)](#): Bitwise OR NOT (shifted register).

[ORR \(immediate\)](#): Bitwise OR (immediate).

[ORR \(shifted register\)](#): Bitwise OR (shifted register).

[PACDA, PACDZA](#): Pointer Authentication Code for Data address, using key A.

[PACDB, PACDZB](#): Pointer Authentication Code for Data address, using key B.

[PACGA](#): Pointer Authentication Code, using Generic key.

[PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA](#): Pointer Authentication Code for Instruction address, using key A.

[PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB](#): Pointer Authentication Code for Instruction address, using key B.

[PRFM \(immediate\)](#): Prefetch Memory (immediate).

[PRFM \(literal\)](#): Prefetch Memory (literal).

[PRFM \(register\)](#): Prefetch Memory (register).

[PRFUM](#): Prefetch Memory (unscaled offset).

[PSB CSYNC](#): Profiling Synchronization Barrier.

[PSSBB](#): Physical Speculative Store Bypass Barrier: an alias of DSB.

[RBIT](#): Reverse Bits.

[RCWCAS, RCWCASA, RCWCASL, RCWCASAL](#): Read Check Write Compare and Swap doubleword in memory.

[RCWCASP, RCWCASPA, RCWCASPL, RCWCASPAL](#): Read Check Write Compare and Swap quadword in memory.

[RCWCLR, RCWCLRA, RCWCLRL, RCWCLRAL](#): Read Check Write atomic bit Clear on doubleword in memory.

[RCWCLRP, RCWCLRPA, RCWCLRPL, RCWCLRPAL](#): Read Check Write atomic bit Clear on quadword in memory.

[RCWSCAS, RCWSCASA, RCWSCASL, RCWSCASAL](#): Read Check Write Software Compare and Swap doubleword in memory.

[RCWSCASP, RCWSCASPA, RCWSCASPL, RCWSCASPAL](#): Read Check Write Software Compare and Swap quadword in memory.

[RCWSCLR, RCWSCLRA, RCWSCLRL, RCWSCLRAL](#): Read Check Write Software atomic bit Clear on doubleword in memory.

[RCWSCLRP, RCWSCLRPA, RCWSCLRPL, RCWSCLRPAL](#): Read Check Write Software atomic bit Clear on quadword in memory.

[RCWSET, RCWSETA, RCWSETL, RCWSETAL](#): Read Check Write atomic bit Set on doubleword in memory.



[RCWSETP, RCWSETPA, RCWSETPL, RCWSETPAL](#): Read Check Write atomic bit Set on quadword in memory.

[RCWSSET, RCWSSETA, RCWSSETL, RCWSSETAL](#): Read Check Write Software atomic bit Set on doubleword in memory.

[RCWSSETP, RCWSSETPA, RCWSSETPL, RCWSSETPAL](#): Read Check Write Software atomic bit Set on quadword in memory.

[RCWSSWP, RCWSSWPA, RCWSSWPL, RCWSSWPAL](#): Read Check Write Software Swap doubleword in memory.

[RCWSSWPE, RCWSSWPPA, RCWSSWPPL, RCWSSWPPLA](#): Read Check Write Software Swap quadword in memory.

[RCWSWP, RCWSWPA, RCWSWPL, RCWSWPAL](#): Read Check Write Swap doubleword in memory.

[RCWSWPE, RCWSWPPA, RCWSWPPL, RCWSWPPLA](#): Read Check Write Swap quadword in memory.

[RET](#): Return from subroutine.

[RETAA, RETAB](#): Return from subroutine, with pointer authentication.

[REV](#): Reverse Bytes.

[REV16](#): Reverse bytes in 16-bit halfwords.

[REV32](#): Reverse bytes in 32-bit words.

[REV64](#): Reverse Bytes: an alias of REV.

[RMIF](#): Rotate, Mask Insert Flags.

[ROR \(immediate\)](#): Rotate right (immediate): an alias of EXTR.

[ROR \(register\)](#): Rotate Right (register): an alias of RORV.

[RORV](#): Rotate Right Variable.

[RPRFM](#): Range Prefetch Memory.

[SB](#): Speculation Barrier.

[SBC](#): Subtract with Carry.

[SBCS](#): Subtract with Carry, setting flags.

[SBFIZ](#): Signed Bitfield Insert in Zero: an alias of SBFM.

[SBFM](#): Signed Bitfield Move.

[SBFX](#): Signed Bitfield Extract: an alias of SBFM.

[SDIV](#): Signed Divide.

[SETF8, SETF16](#): Evaluation of 8 or 16 bit flag values.

[SETGP, SETGM, SETGE](#): Memory Set with tag setting.

[SETGPN, SETGMN, SETGEN](#): Memory Set with tag setting, non-temporal.

[SETGPT, SETGMT, SETGET](#): Memory Set with tag setting, unprivileged.

[SETGPTN, SETGMTN, SETGETN](#): Memory Set with tag setting, unprivileged and non-temporal.

[SETP, SETM, SETE](#): Memory Set.

[SETPN, SETMN, SETEN](#): Memory Set, non-temporal.

[SETPT, SETMT, SETET](#): Memory Set, unprivileged.

[SETPTN, SETMTN, SETETN](#): Memory Set, unprivileged and non-temporal.

[SEV](#): Send Event.

[SEVL](#): Send Event Local.

[SMADDL](#): Signed Multiply-Add Long.

[SMAX \(immediate\)](#): Signed Maximum (immediate).

[SMAX \(register\)](#): Signed Maximum (register).

[SMC](#): Secure Monitor Call.

[SMIN \(immediate\)](#): Signed Minimum (immediate).

[SMIN \(register\)](#): Signed Minimum (register).

[SMNEGL](#): Signed Multiply-Negate Long: an alias of SMSUBL.

[SMSTART](#): Enables access to Streaming SVE mode and SME architectural state: an alias of MSR (immediate).

[SMSTOP](#): Disables access to Streaming SVE mode and SME architectural state: an alias of MSR (immediate).

[SMSUBL](#): Signed Multiply-Subtract Long.

[SMULH](#): Signed Multiply High.

[SMULL](#): Signed Multiply Long: an alias of SMADDL.

[SSBB](#): Speculative Store Bypass Barrier: an alias of DSB.

[ST2G](#): Store Allocation Tags.

[ST64B](#): Single-copy Atomic 64-byte Store without Return.

[ST64BV](#): Single-copy Atomic 64-byte Store with Return.

[ST64BV0](#): Single-copy Atomic 64-byte EL0 Store with Return.

[STADD, STADDL](#): Atomic add on word or doubleword in memory, without return: an alias of LDADD, LDADDA, LDADDAL, LDADDL.

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

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

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

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

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

[STEOR, STEORL](#): Atomic Exclusive-OR on word or doubleword in memory, without return: an alias of LDEOR, LDEORA, LDEORAL, LDEORL.

[STEORB, STEORLB](#): Atomic Exclusive-OR on byte in memory, without return: an alias of LDEORB, LDEORAB, LDEORALB, LDEORLB.

[STEORH, STEORLH](#): Atomic Exclusive-OR on halfword in memory, without return: an alias of LDEORH, LDEORAH, LDEORALH, LDEORLH.

[STG](#): Store Allocation Tag.

[STGM](#): Store Tag Multiple.

[STGP](#): Store Allocation Tag and Pair of registers.

[STILP](#): Store-Release ordered Pair of registers.

[STLLR](#): Store LORelease Register.

[STLLRB](#): Store LORelease Register Byte.

[STLLRH](#): Store LORelease Register Halfword.

[STLR](#): Store-Release Register.

[STLRB](#): Store-Release Register Byte.

[STLRH](#): Store-Release Register Halfword.

[STLUR](#): Store-Release Register (unscaled).

[STLURB](#): Store-Release Register Byte (unscaled).

[STLURH](#): Store-Release Register Halfword (unscaled).

[STLXP](#): Store-Release Exclusive Pair of registers.

[STLXR](#): Store-Release Exclusive Register.

[STLXRB](#): Store-Release Exclusive Register Byte.

[STLXRH](#): Store-Release Exclusive Register Halfword.

[STNP](#): Store Pair of Registers, with non-temporal hint.

[STP](#): Store Pair of Registers.

[STR \(immediate\)](#): Store Register (immediate).

[STR \(register\)](#): Store Register (register).

[STRB \(immediate\)](#): Store Register Byte (immediate).

[STRB \(register\)](#): Store Register Byte (register).

[STRH \(immediate\)](#): Store Register Halfword (immediate).

[STRH \(register\)](#): Store Register Halfword (register).

[STSET, STSETL](#): Atomic bit set on word or doubleword in memory, without return: an alias of LDSET, LDSETA, LDSETAL, LDSETL.

[STSETB, STSETLB](#): Atomic bit set on byte in memory, without return: an alias of LDSETB, LDSETAB, LDSETALB, LDSETLB.

[STSETH, STSETLH](#): Atomic bit set on halfword in memory, without return: an alias of LDSETH, LDSETAH, LDSETALH, LDSETLH.

[STSMAX, STSMAXL](#): Atomic signed maximum on word or doubleword in memory, without return: an alias of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL.

[STSMAXB, STSMAXLB](#): Atomic signed maximum on byte in memory, without return: an alias of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB.

[STSMAXH, STSMAXLH](#): Atomic signed maximum on halfword in memory, without return: an alias of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH.

[STSMIN, STSMINL](#): Atomic signed minimum on word or doubleword in memory, without return: an alias of LDSMIN, LDSMINA, LDSMINAL, LDSMINL.

[STSMINB, STSMINLB](#): Atomic signed minimum on byte in memory, without return: an alias of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB.

[STSMINH, STSMINLH](#): Atomic signed minimum on halfword in memory, without return: an alias of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH.

[STTR](#): Store Register (unprivileged).

[STTRB](#): Store Register Byte (unprivileged).

[STTRH](#): Store Register Halfword (unprivileged).

[STUMAX, STUMAXL](#): Atomic unsigned maximum on word or doubleword in memory, without return: an alias of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL.

[STUMAXB, STUMAXLB](#): Atomic unsigned maximum on byte in memory, without return: an alias of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB.

[STUMAXH, STUMAXLH](#): Atomic unsigned maximum on halfword in memory, without return: an alias of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH.

[STUMIN, STUMINL](#): Atomic unsigned minimum on word or doubleword in memory, without return: an alias of LDUMIN, LDUMINA, LDUMINAL, LDUMINL.

[STUMINB, STUMINLB](#): Atomic unsigned minimum on byte in memory, without return: an alias of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB.

[STUMINH, STUMINLH](#): Atomic unsigned minimum on halfword in memory, without return: an alias of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH.

[STUR](#): Store Register (unscaled).

[STURB](#): Store Register Byte (unscaled).

[STURH](#): Store Register Halfword (unscaled).

[STXP](#): Store Exclusive Pair of registers.

[STXR](#): Store Exclusive Register.

[STXRB](#): Store Exclusive Register Byte.

[STXRH](#): Store Exclusive Register Halfword.

[STZ2G](#): Store Allocation Tags, Zeroing.

[STZG](#): Store Allocation Tag, Zeroing.

[STZGM](#): Store Tag and Zero Multiple.

[SUB \(extended register\)](#): Subtract (extended register).

[SUB \(immediate\)](#): Subtract (immediate).

[SUB \(shifted register\)](#): Subtract (shifted register).

[SUBG](#): Subtract with Tag.

[SUBP](#): Subtract Pointer.

[SUBPS](#): Subtract Pointer, setting Flags.

[SUBS \(extended register\)](#): Subtract (extended register), setting flags.

[SUBS \(immediate\)](#): Subtract (immediate), setting flags.

[SUBS \(shifted register\)](#): Subtract (shifted register), setting flags.

[SVC](#): Supervisor Call.

[SWP, SWPA, SWPAL, SWPL](#): Swap word or doubleword in memory.

[SWPB, SWPAB, SWPALB, SWPLB](#): Swap byte in memory.

[SWPH, SWPAH, SWPALH, SWPLH](#): Swap halfword in memory.

[SWPP, SWPPA, SWPPAL, SWPPL](#): Swap quadword in memory.

[SXTB](#): Signed Extend Byte: an alias of SBFM.

[SXTH](#): Sign Extend Halfword: an alias of SBFM.

[SXTW](#): Sign Extend Word: an alias of SBFM.

[SYS](#): System instruction.

[SYSL](#): System instruction with result.

[SYSP](#): 128-bit System instruction.

[TBNZ](#): Test bit and Branch if Nonzero.

[TBZ](#): Test bit and Branch if Zero.

[TCANCEL](#): Cancel current transaction.

[TCOMMIT](#): Commit current transaction.

[TLBI](#): TLB Invalidate operation: an alias of SYS.

[TLBIP](#): TLB Invalidate Pair operation: an alias of SYSP.

[TRCIT](#): Trace Instrumentation: an alias of SYS.

[TSB CSYNC](#): Trace Synchronization Barrier.

[TST \(immediate\)](#): Test bits (immediate): an alias of ANDS (immediate).

[TST \(shifted register\)](#): Test (shifted register): an alias of ANDS (shifted register).

[TSTART](#): Start transaction.

[TTEST](#): Test transaction state.

[UBFIZ](#): Unsigned Bitfield Insert in Zero: an alias of UBFM.

[UBFM](#): Unsigned Bitfield Move.

[UBFX](#): Unsigned Bitfield Extract: an alias of UBFM.

[UDF](#): Permanently Undefined.

[UDIV](#): Unsigned Divide.

[UMADDL](#): Unsigned Multiply-Add Long.

[UMAX \(immediate\)](#): Unsigned Maximum (immediate).

[UMAX \(register\)](#): Unsigned Maximum (register).

[UMIN \(immediate\)](#): Unsigned Minimum (immediate).

[UMIN \(register\)](#): Unsigned Minimum (register).

[UMNEGL](#): Unsigned Multiply-Negate Long: an alias of UMSUBL.

[UMSUBL](#): Unsigned Multiply-Subtract Long.

[UMULH](#): Unsigned Multiply High.

[UMULL](#): Unsigned Multiply Long: an alias of UMADDL.

[UXTB](#): Unsigned Extend Byte: an alias of UBFM.

[UXTH](#): Unsigned Extend Halfword: an alias of UBFM.

[WFE](#): Wait For Event.

[WFET](#): Wait For Event with Timeout.

[WFI](#): Wait For Interrupt.

[WFIT](#): Wait For Interrupt with Timeout.

[XAFLAG](#): Convert floating-point condition flags from external format to Arm format.

[XPACD](#), [XPACL](#), [XPACLR](#): Strip Pointer Authentication Code.

[YIELD](#): YIELD.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# ABS

Absolute value computes the absolute value of the signed integer value in the source register, and writes the result to the destination register.

## Integer (FEAT\_CSSC)

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

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

ABS <Wd>, <Wn>

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

ABS <Xd>, <Xn>

```
if !HaveCSSC() then UNDEFINED;
integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer d = UInt(Rd);
```

## Assembler Symbols

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

## Operation

```
bits(datasize) operand1 = X[n, datasize];
integer result = Abs(SInt(operand1));
X[d, datasize] = result<datasize-1:0>;
```

## Operational information

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

ADC

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

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

op S

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

Operation

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

(result, -) = AddWithCarry(operand1, operand2, PSTATE.C);

X[d, datasize] = result;
```

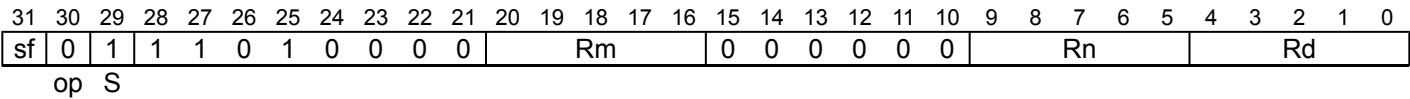
Operational information

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



ADCS

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
bits(4) nzcvc;

(result, nzcvc) = AddWithCarry(operand1, operand2, PSTATE.C);

PSTATE.<N,Z,C,V> = nzcvc;

X[d, datasize] = result;
```

Operational information

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

ADD (extended register)

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

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

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

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

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

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

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

option	<extend>
000	UXTB
001	UXTH
010	LSL UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

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

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

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

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

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

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[<datasize-1:0>] else X[n, datasize];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);

(result, -) = AddWithCarry(operand1, operand2, '0');

if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

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

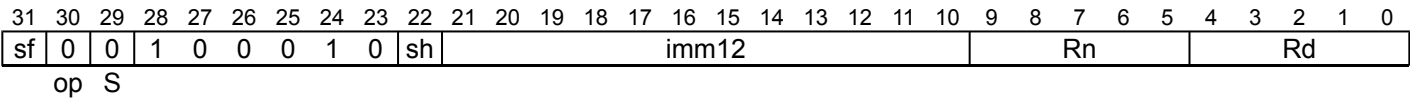
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD (immediate)

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

This instruction is used by the alias [MOV \(to/from SP\)](#).



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

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

Assembler Symbols

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

sh	<shift>
0	LSL #0
1	LSL #12

Alias Conditions

Alias	Is preferred when
<a href="#">MOV (to/from SP)</a>	sh == '0' && imm12 == '0000000000000' && (Rd == '11111'    Rn == '11111')

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[<datasize-1:0>] else X[n, datasize];

(result, -) = AddWithCarry(operand1, imm, '0');

if d == 31 then
  SP[] = ZeroExtend(result, 64);
else
  X[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

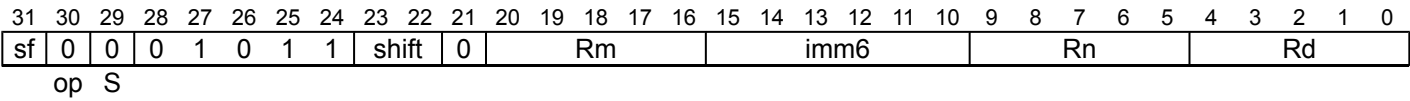
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD (shifted register)

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

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

Assembler Symbols

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

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

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

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

Operation

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

(result, -) = AddWithCarry(operand1, operand2, '0');
X[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

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

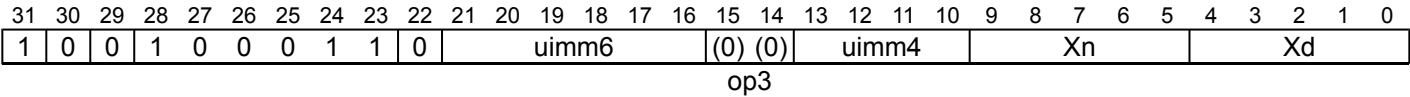
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADDG

Add with Tag adds an immediate value scaled by the Tag granule to the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR\_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

Integer  
(FEAT\_MTE)



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

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

Assembler Symbols

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

Operation

```
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(4) start_tag = AArch64.AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;

if AArch64.AllocationTagAccessIsEnabled(PSTATE.EL) then
    rtag = AArch64.ChooseNonExcludedTag(start_tag, uimm4, exclude);
else
    rtag = '0000';

(result, -) = AddWithCarry(operand1, offset, '0');
result = AArch64.AddressWithAllocationTag(result, rtag);

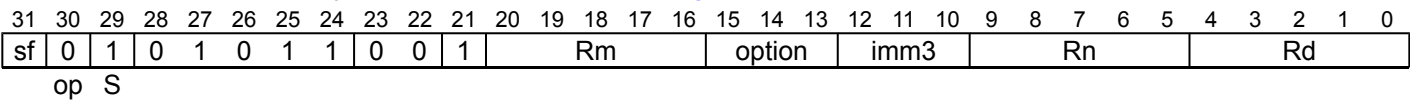
if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;
```



ADDS (extended register)

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

This instruction is used by the alias [CMN \(extended register\)](#).



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

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

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

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

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

option	<extend>
000	UXTB
001	UXTH
010	LSL UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

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

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

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

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

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

Alias Conditions

Alias	Is preferred when
<a href="#">CMN (extended register)</a>	Rd == '11111'

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);
bits(4) nzcvc;

(result, nzcvc) = AddWithCarry(operand1, operand2, '0');

PSTATE.<N,Z,C,V> = nzcvc;

X[d, datasize] = result;
```

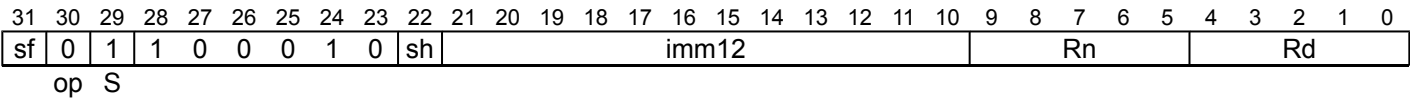
Operational information

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

ADDS (immediate)

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

This instruction is used by the alias [CMN \(immediate\)](#).



## Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADDS (shifted register)

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

This instruction is used by the alias [CMN \(shifted register\)](#).

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

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

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

Assembler Symbols

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

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

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

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

Alias Conditions

Alias	Is preferred when
<a href="#">CMN (shifted register)</a>	Rd == '11111'

## Operation

```
bits(datasize) result;  
bits(datasize) operand1 = X[n, datasize];  
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);  
bits(4) nzcvc;  
  
(result, nzcvc) = AddWithCarry(operand1, operand2, '0');  
  
PSTATE.<N,Z,C,V> = nzcvc;  
  
X[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

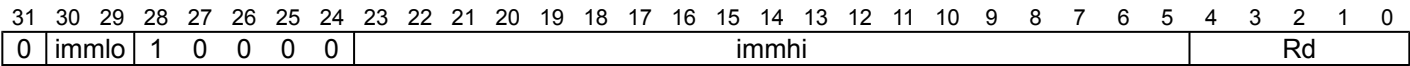
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# ADR

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



op

ADR <Xd>, <label>

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

imm = SignExtend(immhi:immlo, 64);
```

## Assembler Symbols

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

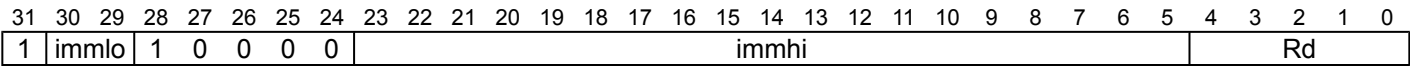
## Operation

```
bits(64) base = PC[];

X[d, 64] = base + imm;
```

ADRP

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



op

```
ADRP <Xd>, <label>
```

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

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

Assembler Symbols

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

Operation

```
bits(64) base = PC[];

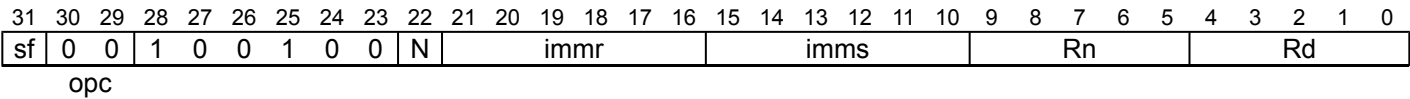
base<11:0> = Zeros(12);

X[d, 64] = base + imm;
```



## AND (immediate)

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



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

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

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

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

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

## Assembler Symbols

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

## Operation

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

result = operand1 AND imm;
if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

## Operational information

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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## AND (shifted register)

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

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

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

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

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

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

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

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

## Assembler Symbols

- <Wd>

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

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

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

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

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

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

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR
- <amount>

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

## Operation

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

result = operand1 AND operand2;
X[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ANDS (immediate)

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

This instruction is used by the alias [TST \(immediate\)](#).

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

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

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

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

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

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

bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE, datasize);
```

## Assembler Symbols

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

## Alias Conditions

Alias	Is preferred when
<a href="#">TST (immediate)</a>	Rd == '11111'

## Operation

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

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

X[d, datasize] = result;
```

## Operational information

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



ANDS (shifted register)

Bitwise AND (shifted register), setting flags, performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result. This instruction is used by the alias [TST \(shifted register\)](#).

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

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

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

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

Assembler Symbols

- <Wd>

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

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

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

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

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

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

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR
- <amount>

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

Alias Conditions

Alias	Is preferred when
<a href="#">TST (shifted register)</a>	Rd == '11111'

## Operation

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

result = operand1 AND operand2;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASR (immediate)

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

This is an alias of [SBFM](#). This means:

- The encodings in this description are named to match the encodings of [SBFM](#).
- The description of [SBFM](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

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

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

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

is equivalent to

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

and is always the preferred disassembly.

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

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

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

Operation

The description of [SBFM](#) gives the operational pseudocode for this instruction.

Operational information

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



## ASR (register)

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

This is an alias of [ASRV](#). This means:

- The encodings in this description are named to match the encodings of [ASRV](#).
- The description of [ASRV](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

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

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

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

is equivalent to

[ASRV](#) <Wd>, <Wn>, <Wm>

and is always the preferred disassembly.

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

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

is equivalent to

[ASRV](#) <Xd>, <Xn>, <Xm>

and is always the preferred disassembly.

## Assembler Symbols

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

## Operation

The description of [ASRV](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

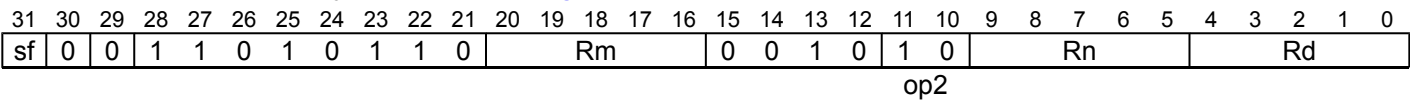
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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



ASRV

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

This instruction is used by the alias [ASR \(register\)](#).



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m, datasize];

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

Operational information

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

AT

Address Translate. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions*.

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

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

AT [<at\\_op>](#), [<Xt>](#)

is equivalent to

[SYS](#) [#<op1>](#), C7, [<Cm>](#), [#<op2>](#), [<Xt>](#)

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

Assembler Symbols

[<at\\_op>](#) Is an AT instruction name, as listed for the AT system instruction group, encoded in “[op1](#):[CRm](#)<0>:[op2](#)”:

op1	CRm<0>	op2	<at_op>	Architectural Feature
000	0	000	S1E1R	-
000	0	001	S1E1W	-
000	0	010	S1E0R	-
000	0	011	S1E0W	-
000	1	000	S1E1RP	FEAT_PAN2
000	1	001	S1E1WP	FEAT_PAN2
000	1	010	S1E1A	FEAT_ATS1A
100	0	000	S1E2R	-
100	0	001	S1E2W	-
100	0	100	S12E1R	-
100	0	101	S12E1W	-
100	0	110	S12E0R	-
100	0	111	S12E0W	-
100	1	010	S1E2A	FEAT_ATS1A
110	0	000	S1E3R	-
110	0	001	S1E3W	-
110	1	010	S1E3A	FEAT_ATS1A

- [<op1>](#) Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “[op1](#)” field.
- [<Cm>](#) Is a name ‘Cm’, with ‘m’ in the range 0 to 15, encoded in the “[CRm](#)” field.
- [<op2>](#) Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “[op2](#)” field.
- [<Xt>](#) Is the 64-bit name of the general-purpose source register, encoded in the “[Rt](#)” field.

Operation

The description of [SYS](#) gives the operational pseudocode for this instruction.

AUTDA, AUTDZA

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

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

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see *Faulting on pointer authentication*.

Integer  
(FEAT\_PAuth)

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

AUTDA (Z == 0)

```
AUTDA <Xd>, <Xn|SP>
```

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

```
AUTDZA <Xd>

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

if !HavePACExt() then
    UNDEFINED;

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

Assembler Symbols

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

Operation

```
if HavePACExt() then
    if source_is_sp then
        X[d, 64] = AuthDA(X[d, 64], SP[], FALSE);
    else
        X[d, 64] = AuthDA(X[d, 64], X[n, 64], FALSE);
```

## AUTDB, AUTDZB

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

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

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see [Faulting on pointer authentication](#).

### Integer (FEAT\_PAuth)

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

### AUTDB (Z == 0)

AUTDB <Xd>, <Xn|SP>

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

```
AUTDZB <Xd>

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

if !HavePACExt() then
    UNDEFINED;

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

### Assembler Symbols

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

### Operation

```
if HavePACExt() then
    if source_is_sp then
        X[d, 64] = AuthDB(X[d, 64], SP[], FALSE);
    else
        X[d, 64] = AuthDB(X[d, 64], X[n, 64], FALSE);
```

AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA

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

The address is:

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

The modifier is:

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

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see *Faulting on pointer authentication*.

It has encodings from 2 classes: Integer and System

Integer
(FEAT\_PAuth)

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

AUTIA (Z == 0)

AUTIA <Xd>, <Xn|SP>

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

AUTIZA <Xd>

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

if !HavePACExt() then
    UNDEFINED;

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

System
(FEAT\_PAuth)

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

## AUTIA1716 (CRm == 0001 && op2 == 100)

AUTIA1716

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

AUTIASP

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

AUTIAZ

```
integer d;
integer n;
boolean source_is_sp = FALSE;

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

## Assembler Symbols

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

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

## Operation

```
if HavePACExt() then
  if source_is_sp then
    X[d, 64] = AuthIA(X[d, 64], SP[], FALSE);
  else
    X[d, 64] = AuthIA(X[d, 64], X[n, 64], FALSE);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB

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

The address is:

- In the general-purpose register that is specified by <Xd> for AUTIB and AUTIZB.
- In X17, for AUTIB1716.
- In X30, for AUTIBSP and AUTIBZ.

The modifier is:

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

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see *Faulting on pointer authentication*.

It has encodings from 2 classes: Integer and System

Integer

(FEAT\_PAuth)

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

AUTIB (Z == 0)

AUTIB <Xd>, <Xn|SP>

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

AUTIZB <Xd>

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

if !HavePACExt() then
    UNDEFINED;

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

System

(FEAT\_PAuth)

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

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

AUTIB1716

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

AUTIBSP

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

AUTIBZ

```
integer d;
integer n;
boolean source_is_sp = FALSE;

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

## Assembler Symbols

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

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

## Operation

```
if HavePACExt() then
  if source_is_sp then
    X[d, 64] = AuthIB(X[d, 64], SP[], FALSE);
  else
    X[d, 64] = AuthIB(X[d, 64], X[n, 64], FALSE);
```

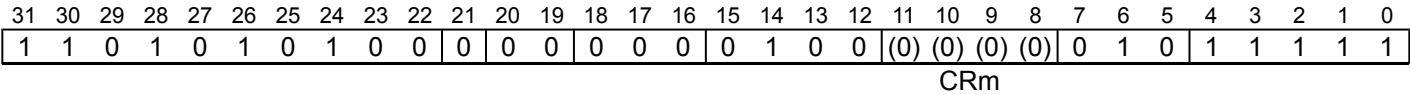
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AXFLAG

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

System  
(FEAT\_FlagM2)



AXFLAG

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

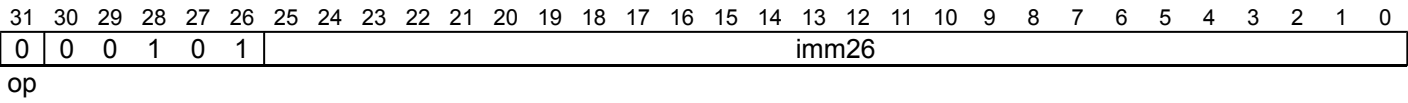
Operation

```
bit z = PSTATE.Z OR PSTATE.V;
bit c = PSTATE.C AND NOT(PSTATE.V);

PSTATE.N = '0';
PSTATE.Z = z;
PSTATE.C = c;
PSTATE.V = '0';
```

B

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



B <label>

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

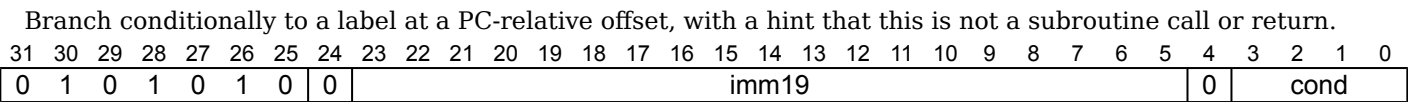
Assembler Symbols

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

Operation

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

B.cond



B.<cond> <label>

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

Assembler Symbols

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

Operation

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

# BC.cond

Branch Consistent conditionally to a label at a PC-relative offset, with a hint that this branch will behave very consistently and is very unlikely to change direction.

## 19-bit signed PC-relative branch offset (FEAT\_HBC)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	0	imm19																		1	cond				

BC.<cond> <label>

```
if !HaveFeatHBC() then UNDEFINED;
bits(64) offset = SignExtend(imm19:'00', 64);
```

## Assembler Symbols

- <cond>

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

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

## Operation

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

# BFC

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

This is an alias of [BFM](#). This means:

- The encodings in this description are named to match the encodings of [BFM](#).
- The description of [BFM](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

## Leaving other bits unchanged (FEAT\_ASMv8p2)

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

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

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

is equivalent to

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

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

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

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

is equivalent to

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

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

## Assembler Symbols

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

## Operation

The description of [BFM](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

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

- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



BFI

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

This is an alias of [BFM](#). This means:

- The encodings in this description are named to match the encodings of [BFM](#).
- The description of [BFM](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		0	1	1	0	0	1	1	0	N	immr						imms						!= 11111				Rd				
opc											Rn																				

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

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

is equivalent to

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

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

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

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

is equivalent to

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

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

Assembler Symbols

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

Operation

The description of [BFM](#) gives the operational pseudocode for this instruction.

Operational information

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



BFM

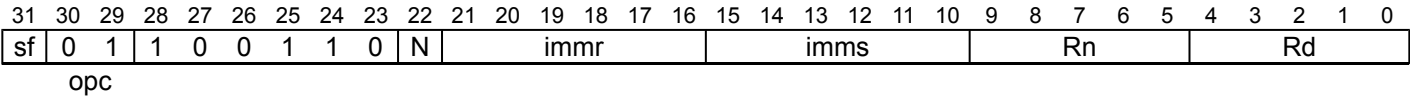
Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

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

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

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

This instruction is used by the aliases [BFC](#), [BFI](#), and [BFXIL](#).



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

BFM <Wd>, <Wn>, #<immr>, #<imms>

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

BFM <Xd>, <Xn>, #<immr>, #<imms>

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

integer r;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;

r = UInt(immr);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE, datasize);
```

Assembler Symbols

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

Alias Conditions

Alias	Is preferred when
BFC	Rn == '11111' && UInt(imms) < UInt(immr)
BFI	Rn != '11111' && UInt(imms) < UInt(immr)
BFXIL	UInt(imms) >= UInt(immr)

## Operation

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

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

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

## Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BFXIL

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

This is an alias of [BFM](#). This means:

- The encodings in this description are named to match the encodings of [BFM](#).
- The description of [BFM](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

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

opc

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

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

is equivalent to

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

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

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

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

is equivalent to

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

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

## Assembler Symbols

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

## Operation

The description of [BFM](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

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



## BIC (shifted register)

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

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

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

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

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

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

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

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

## Assembler Symbols

- <Wd>

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

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

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

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

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

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

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR
- <amount>

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

## Operation

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

operand2 = NOT(operand2);

result = operand1 AND operand2;
X[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

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

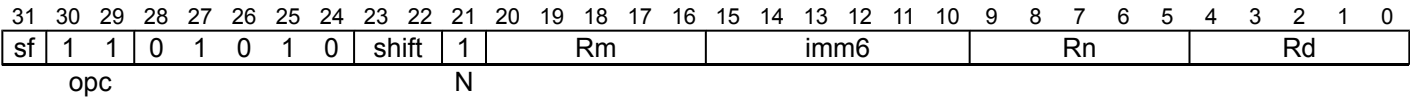
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



BICS (shifted register)

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

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

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

Assembler Symbols

- <Wd>

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

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

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

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

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

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

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR
- <amount>

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

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

## Operation

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

operand2 = NOT(operand2);

result = operand1 AND operand2;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

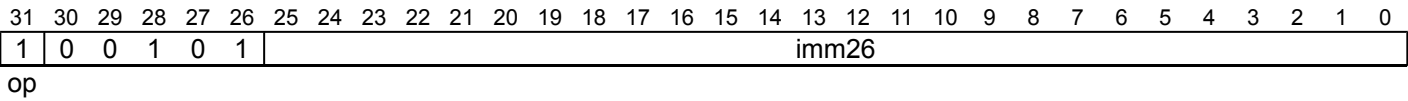
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BL

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



BL <label>

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

Assembler Symbols

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

Operation

```
if HaveGCS() && GCSPCREnabled(PSTATE.EL) then
    AddGCSRecord(PC[] + 4);
X[30, 64] = PC[] + 4;
BranchTo(PC[] + offset, BranchType_DIRCALL, FALSE);
```

# BLR

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

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

BLR <Xn>

```
integer n = UInt(Rn);
```

## Assembler Symbols

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

## Operation

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

if HaveGCS() && GCSPCREnabled(PSTATE.EL) then
    AddGCSRecord(PC[] + 4);
X[30, 64] = PC[] + 4;

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

# BLRAA, BLRAAZ, BLRAB, BLRABZ

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

The modifier is:

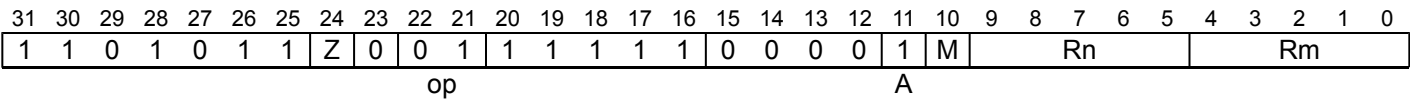
- In the general-purpose register or stack pointer that is specified by <Xm|SP> for BLRAA and BLRAB.
- The value zero, for BLRAAZ and BLRABZ.

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

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see *Faulting on pointer authentication*.

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

## Integer (FEAT\_PAuth)



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

```
BLRAAZ <Xn>
```

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

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

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

```
BLRABZ <Xn>
```

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

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

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

if !HavePACExt() then
    UNDEFINED;

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

## Assembler Symbols

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

## Operation

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

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

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

if HaveGCS() && GCSPCREnabled(PSTATE.EL) then
    AddGCSRecord(PC[] + 4);
X[30, 64] = PC[] + 4;

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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



BRAA, BRAAZ, BRAB, BRABZ

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

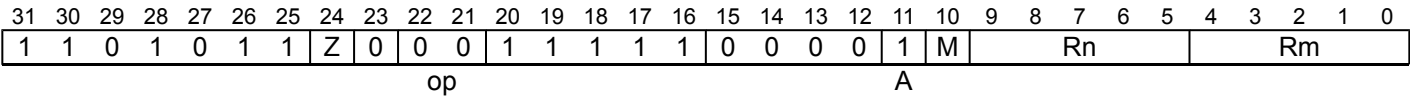
- In the general-purpose register or stack pointer that is specified by <Xm|SP> for BRAA and BRAB.
- The value zero, for BRAAZ and BRABZ.

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

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see *Faulting on pointer authentication*.

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

Integer  
(FEAT\_PAuth)



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

```
BRAAZ <Xn>
```

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

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

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

```
BRABZ <Xn>
```

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

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

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

if !HavePACExt() then
    UNDEFINED;

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

Assembler Symbols

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



## Operation

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

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

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

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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

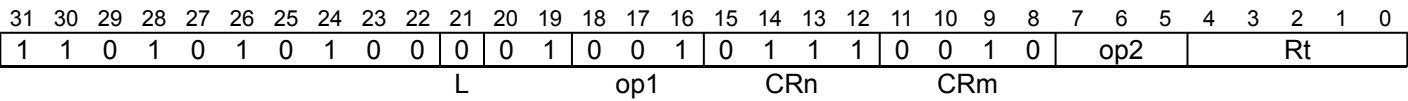
BRB

Branch Record Buffer. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions*.

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System  
(FEAT\_BRBE)



BRB <brb\_op>{, <Xt>}

is equivalent to

[SYS](#) #1, C7, C2, #<op2>{, <Xt>}

and is the preferred disassembly when `SysOp('001', '0111', '0010', op2) == Sys_BRB`.

Assembler Symbols

<brb\_op> Is a BRB instruction name, as listed for the BRB system instruction group, encoded in “op2”:

op2	<brb_op>	Architectural Feature
100	IALL	FEAT_BRBE
101	INJ	FEAT_BRBE

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

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

The description of [SYS](#) gives the operational pseudocode for this instruction.

# BRK

Breakpoint instruction. A BRK instruction generates a Breakpoint Instruction exception. The PE records the exception in *ESR\_ELx*, using the EC value 0x3c, and captures the value of the immediate argument in *ESR\_ELx*.ISS.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	0	0	1	imm16																0	0	0	0	0

BRK #<imm>

```
if HaveBTIExt() then
    SetBTypeCompatible(TRUE);
```

## Assembler Symbols

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

## Operation

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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BTI

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

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

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

## Note

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

## System (FEAT\_BTI)

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

BTI {<targets>}

```
SystemHintOp op;

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

## Assembler Symbols

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

op2<2:1>	<targets>
00	(omitted)
01	c
10	j
11	jc

## Operation

```
case op of
  when SystemHintOp_YIELD
    Hint_Yield();

  when SystemHintOp_DGH
    Hint_DGH();

  when SystemHintOp_WFE
    integer localtimeout = 1 << 64;    // No local timeout event is generated
    Hint_WFE(localtimeout, WfxType_WFE);

  when SystemHintOp_WFI
    integer localtimeout = 1 << 64;    // No local timeout event is generated
    Hint_WFI(localtimeout, WfxType_WFI);

  when SystemHintOp_SEV
    SendEvent();

  when SystemHintOp_SEVL
    SendEventLocal();

  when SystemHintOp_ESB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure_ERR, FALSE);
      SynchronizeErrors();
      AArch64.ESBOperation();
      if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESBOperation();
      TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();

  when SystemHintOp_TSB
    TraceSynchronizationBarrier();

  when SystemHintOp_GCSB
    GCSSynchronizationBarrier();

  when SystemHintOp_CHKFEAT
    X[16, 64] = AArch64.ChkFeat(X[16, 64]);

  when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();

  when SystemHintOp_CLRBHB
    Hint_CLRBHB();

  when SystemHintOp_BTI
    SetBTypeNext('00');

  when SystemHintOp_NOP
    return;    // do nothing

  otherwise
    Unreachable();
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CAS, CASA, CASAL, CASL

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

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

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

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

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

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

No offset  
(FEAT\_LSE)

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

size

### 32-bit CAS (size == 10 && L == 0 && o0 == 0)

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

### 32-bit CASA (size == 10 && L == 1 && o0 == 0)

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

### 32-bit CASAL (size == 10 && L == 1 && o0 == 1)

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

### 32-bit CASL (size == 10 && L == 0 && o0 == 1)

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

### 64-bit CAS (size == 11 && L == 0 && o0 == 0)

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

### 64-bit CASA (size == 11 && L == 1 && o0 == 0)

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

### 64-bit CASAL (size == 11 && L == 1 && o0 == 1)

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

### 64-bit CASL (size == 11 && L == 0 && o0 == 1)

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

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = L == '1';
boolean release = o0 == '1';
boolean tagchecked = n != 31;
```

## Assembler Symbols

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

## Operation

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

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release, tagchecked);

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

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, comparevalue, newvalue, accdesc);

X[s, regsize] = ZeroExtend(data, regsize);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# CASB, CASAB, CASALB, CASLB

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

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

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

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

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

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

## No offset (FEAT\_LSE)

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

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

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

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

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

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

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

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

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

```
if !HaveAtomicExt() then UNDEFINED;

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

boolean acquire = L == '1';
boolean release = o0 == '1';
boolean tagchecked = n != 31;
```

## Assembler Symbols

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

## Operation

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

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release, tagchecked);

comparevalue = X[s, 8];
newvalue = X[t, 8];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, comparevalue, newvalue, accdesc);

X[s, 32] = ZeroExtend(data, 32);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# CASH, CASAH, CASALH, CASLH

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

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

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

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

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

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

## No offset (FEAT\_LSE)

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

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

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

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

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

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

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

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

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

```
if !HaveAtomicExt() then UNDEFINED;

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

boolean acquire = L == '1';
boolean release = o0 == '1';
boolean tagchecked = n != 31;
```

## Assembler Symbols

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

## Operation

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

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release, tagchecked);

comparevalue = X[s, 16];
newvalue = X[t, 16];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, comparevalue, newvalue, accdesc);

X[s, 32] = ZeroExtend(data, 32);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CASP, CASPA, CASPAL, CASPL

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

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

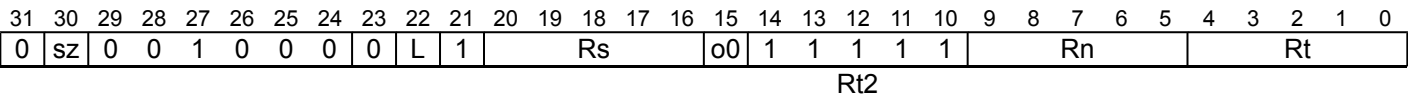
For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

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

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

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

No offset  
(FEAT\_LSE)



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
if !HaveAtomicExt() then UNDEFINED;
if Rs<0> == '1' then UNDEFINED;
if Rt<0> == '1' then UNDEFINED;

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

integer datasize = 32 << UInt(sz);
boolean acquire = L == '1';
boolean release = o0 == '1';
boolean tagchecked = n != 31;
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Ws> must be an even-numbered register.
<W(s+1)>	Is the 32-bit name of the second general-purpose register to be compared and loaded.
<Wt>	Is the 32-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Wt> must be an even-numbered register.
<W(t+1)>	Is the 32-bit name of the second general-purpose register to be conditionally stored.
<Xs>	Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Xs> must be an even-numbered register.
<X(s+1)>	Is the 64-bit name of the second general-purpose register to be compared and loaded.
<Xt>	Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Xt> must be an even-numbered register.

- <X(t+1)> Is the 64-bit name of the second general-purpose register to be conditionally stored.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```

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

bits(datasize) s1 = X[s, datasize];
bits(datasize) s2 = X[s+1, datasize];
bits(datasize) t1 = X[t, datasize];
bits(datasize) t2 = X[t+1, datasize];

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release, tagchecked);

comparevalue = if BigEndian(accdesc.acctype) then s1:s2 else s2:s1;
newvalue = if BigEndian(accdesc.acctype) then t1:t2 else t2:t1;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, comparevalue, newvalue, accdesc);

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

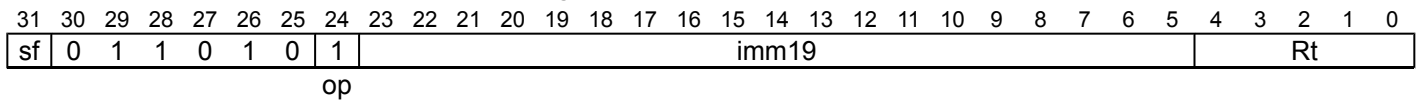
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

**CBNZ**

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



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

CBNZ <Wt>, <label>

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

CBNZ <Xt>, <label>

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

## Assembler Symbols

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

## Operation

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

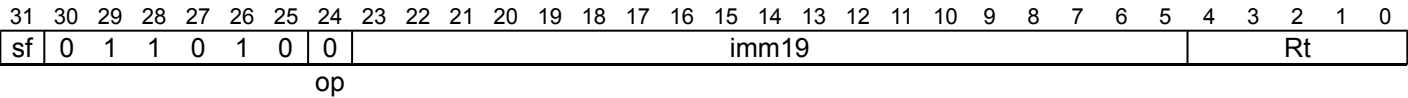
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



CBZ

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



32-bit (sf == 0)

```
CBZ <Wt>, <label>
```

64-bit (sf == 1)

```
CBZ <Xt>, <label>
```

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

Assembler Symbols

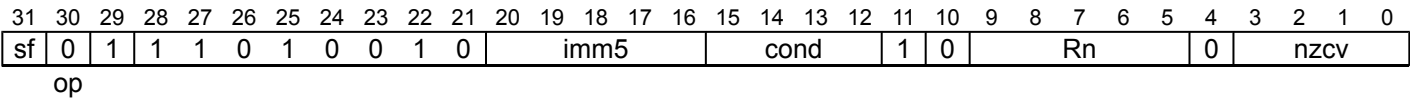
- <Wt> Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- <label> Is the program label to be conditionally branched to. Its offset from the address of this instruction,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

Operation

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

CCMN (immediate)

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



32-bit (sf == 0)

CCMN <Wn>, #<imm>, #<nzcw>, <cond>

64-bit (sf == 1)

CCMN <Xn>, #<imm>, #<nzcw>, <cond>

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

Assembler Symbols

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

Operation

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

Operational information

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

## CCMN (register)

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

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

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

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

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

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

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

## Assembler Symbols

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

## Operation

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

## Operational information

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

CCMP (immediate)

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		1	1	1	1	0	1	0	0	1	0	imm5					cond				1	0	Rn				0	nzcw			

op

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

Operation

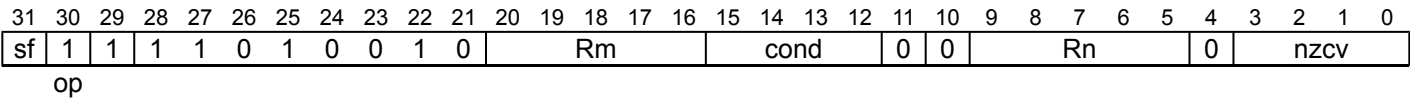
```
if ConditionHolds(cond) then
    bits(datasize) operand1 = X[n, datasize];
    bits(datasize) operand2;
    operand2 = NOT(imm);
    (-, flags) = AddWithCarry(operand1, operand2, '1');
PSTATE.<N,Z,C,V> = flags;
```

Operational information

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

CCMP (register)

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



32-bit (sf == 0)

```
CCMP <Wn>, <Wm>, #<nzcw>, <cond>
```

64-bit (sf == 1)

```
CCMP <Xn>, <Xm>, #<nzcw>, <cond>
```

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

Assembler Symbols

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

Operation

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

Operational information

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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CFINV

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

System  
(FEAT\_FlagM)

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

CRm

CFINV

```
if !HaveFlagManipulateExt() then UNDEFINED;
```

Operation

```
PSTATE.C = NOT(PSTATE.C);
```

Operational information

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

CFP

Control Flow Prediction Restriction by Context prevents control flow predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. Control flow predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot be used to exploitatively control speculative execution occurring after the instruction is complete and synchronized.

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

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System  
(FEAT\_SPECRES)

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

CFP RCTX, [<Xt>](#)

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

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

Operation

The description of [SYS](#) gives the operational pseudocode for this instruction.

# CHKFEAT

Check feature status. This instruction indicates the status of features.  
If FEAT\_CHK is not implemented, this instruction executes as a NOP.

## System (FEAT\_CHK)

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

### CHKFEAT X16

```
if !IsFeatureImplemented(FEAT_CHK) then EndOfInstruction();
```

## Operation

```
X[16, 64] = AArch64.ChkFeat(X[16, 64]);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42  
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# CINC

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

This is an alias of [CSINC](#). This means:

- The encodings in this description are named to match the encodings of [CSINC](#).
- The description of [CSINC](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	0	0	!= 11111				!= 111x				0	1	!= 11111				Rd						
op											Rm				cond				o2		Rn										

## 32-bit (sf == 0)

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

is equivalent to

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

and is the preferred disassembly when `Rn == Rm`.

## 64-bit (sf == 1)

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

is equivalent to

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

and is the preferred disassembly when `Rn == Rm`.

## Assembler Symbols

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

## Operation

The description of [CSINC](#) gives the operational pseudocode for this instruction.

## Operational information

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

CINV

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

This is an alias of CSINV. This means:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	1	1	0	1	0	1	0	0	!= 11111				!= 111x				0	0	!= 11111				Rd						
op											Rm				cond				o2		Rn										

32-bit (sf == 0)

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

is equivalent to

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

and is the preferred disassembly when Rn == Rm.

64-bit (sf == 1)

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

is equivalent to

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

and is the preferred disassembly when Rn == Rm.

Assembler Symbols

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

Operation

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

Operational information

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

# CLRBHB

Clear Branch History clears the branch history for the current context to the extent that branch history information created before the CLRBHB instruction cannot be used by code before the CLRBHB instruction to exploitatively control the execution of any indirect branches in code in the current context that appear in program order after the instruction.

## System (FEAT\_CLRBHB)

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

## CLRBHB

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

## Operation

```
Hint_CLRBHB();
```

# CLREX

Clear Exclusive clears the local monitor of the executing PE.

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

CLREX {#<imm>}

// CRm field is ignored

## Assembler Symbols

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

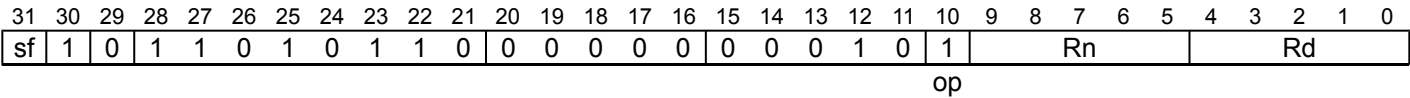
## Operation

[ClearExclusiveLocal](#) ([ProcessorID](#)());

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42  
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CLS

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



32-bit (sf == 0)

```
CLS <Wd>, <Wn>
```

64-bit (sf == 1)

```
CLS <Xd>, <Xn>
```

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

Assembler Symbols

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

Operation

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

result = CountLeadingSignBits(operand1);

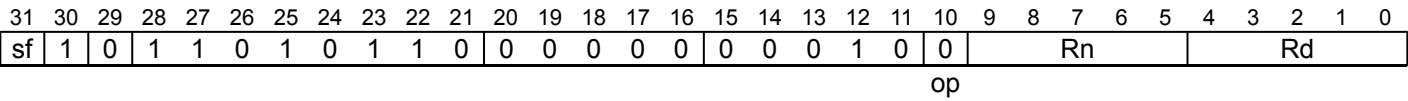
X[d, datasize] = result<datasize-1:0>;
```

Operational information

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

# CLZ

Count Leading Zeros counts the number of consecutive binary zero bits, starting from the most significant bit in the source register, and places the count in the destination register.



32-bit (sf == 0)

```
CLZ <Wd>, <Wn>
```

64-bit (sf == 1)

```
CLZ <Xd>, <Xn>
```

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

## Assembler Symbols

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

## Operation

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

result = CountLeadingZeroBits(operand1);
X[d, datasize] = result<datasize-1:0>;
```

## Operational information

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

CMN (extended register)

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

This is an alias of [ADDS \(extended register\)](#). This means:

- The encodings in this description are named to match the encodings of [ADDS \(extended register\)](#).
- The description of [ADDS \(extended register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

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

32-bit (sf == 0)

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

is equivalent to

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

and is always the preferred disassembly.

64-bit (sf == 1)

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

- <Wn|WSP>

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

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

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

<R>

Is a width specifier, encoded in "option":

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

- <m>

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

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

option	<extend>
000	UXTB
001	UXTH
010	LSL   UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

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

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

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL   UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

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

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

## Operation

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

## Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



CMN (immediate)

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

This is an alias of [ADDS \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [ADDS \(immediate\)](#).
  - The description of [ADDS \(immediate\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.
- |    |    |    |    |    |    |    |    |    |    |    |    |       |    |    |    |    |    |    |    |    |    |    |   |   |           |   |   |   |   |   |   |
|----|----|----|----|----|----|----|----|----|----|----|----|-------|----|----|----|----|----|----|----|----|----|----|---|---|-----------|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19    | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8 | 7 | 6         | 5 | 4 | 3 | 2 | 1 | 0 |
| sf |    | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | sh |    | imm12 |    |    |    |    |    |    |    |    |    | Rn |   |   | 1 1 1 1 1 |   |   |   |   |   |   |
| op |    |    |    |    |    |    |    |    |    | S  |    |       |    |    |    |    |    |    |    | Rd |    |    |   |   |           |   |   |   |   |   |   |

32-bit (sf == 0)

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

is equivalent to

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

and is always the preferred disassembly.

64-bit (sf == 1)

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

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

sh	<shift>
0	LSL #0
1	LSL #12

Operation

The description of [ADDS \(immediate\)](#) gives the operational pseudocode for this instruction.

Operational information

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



CMN (shifted register)

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

This is an alias of [ADDS \(shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [ADDS \(shifted register\)](#).
- The description of [ADDS \(shifted register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

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

32-bit (sf == 0)

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

is equivalent to

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

and is always the preferred disassembly.

64-bit (sf == 1)

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

- <Wn>

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

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

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

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

<shift>

Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

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

<amount>

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

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

Operation

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

## Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMP (extended register)

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

This is an alias of [SUBS \(extended register\)](#). This means:

- The encodings in this description are named to match the encodings of [SUBS \(extended register\)](#).
- The description of [SUBS \(extended register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

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

32-bit (sf == 0)

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

is equivalent to

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

and is always the preferred disassembly.

64-bit (sf == 1)

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

- <Wn|WSP>

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

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

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

Is a width specifier, encoded in "option":

option	<R>
00x	W
010	W
x11	X
10x	W
110	W
- <m>

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

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

option	<extend>
000	UXTB
001	UXTH
010	LSL   UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

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

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

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL   UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

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

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

## Operation

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

## Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMP (immediate)

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

This is an alias of [SUBS \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [SUBS \(immediate\)](#).
- The description of [SUBS \(immediate\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

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

32-bit (sf == 0)

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

is equivalent to

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

and is always the preferred disassembly.

64-bit (sf == 1)

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

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

sh	<shift>
0	LSL #0
1	LSL #12

Operation

The description of [SUBS \(immediate\)](#) gives the operational pseudocode for this instruction.

Operational information

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



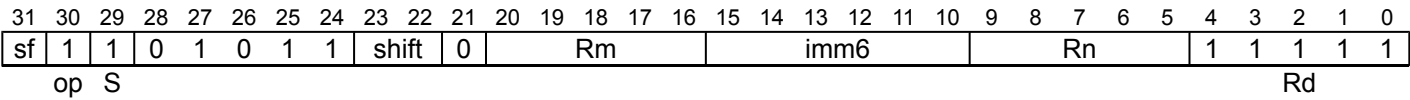


CMP (shifted register)

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

This is an alias of [SUBS \(shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [SUBS \(shifted register\)](#).
- The description of [SUBS \(shifted register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit (sf == 0)

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

is equivalent to

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

and is always the preferred disassembly.

64-bit (sf == 1)

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

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

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

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

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

Operation

The description of [SUBS \(shifted register\)](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

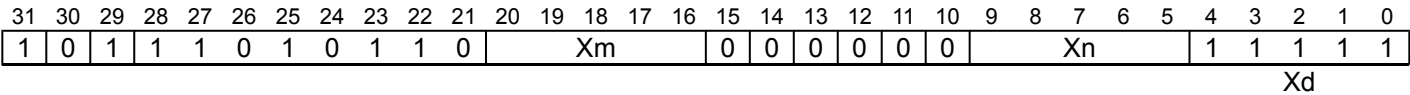
CMPP

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

This is an alias of [SUBPS](#). This means:

- The encodings in this description are named to match the encodings of [SUBPS](#).
- The description of [SUBPS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer  
(FEAT\_MTE)



CMPP <Xn | SP>, <Xm | SP>

is equivalent to

[SUBPS](#) XZR, <Xn | SP>, <Xm | SP>

and is always the preferred disassembly.

Assembler Symbols

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

Operation

The description of [SUBPS](#) gives the operational pseudocode for this instruction.

CNEG

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

This is an alias of CSNEG. This means:

- The encodings in this description are named to match the encodings of CSNEG.
- The description of CSNEG gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sf		1		0		1		1		0		1		0		1		0		0		Rm						!= 111x		0		1		Rn						Rd	
op																		cond						o2																	

32-bit (sf == 0)

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

is equivalent to

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

and is the preferred disassembly when Rn == Rm.

64-bit (sf == 1)

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

is equivalent to

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

and is the preferred disassembly when Rn == Rm.

Assembler Symbols

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

Operation

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

Operational information

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

# CNT

Count bits counts the number of binary one bits in the value of the source register, and writes the result to the destination register.

## Integer (FEAT\_CSSC)

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

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

CNT <Wd>, <Wn>

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

CNT <Xd>, <Xn>

```
if !HaveCSSC() then UNDEFINED;
integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer d = UInt(Rd);
```

## Assembler Symbols

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

## Operation

```
bits(datasize) operand1 = X[n, datasize];
integer result = BitCount(operand1);
X[d, datasize] = result<datasize-1:0>;
```

## Operational information

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

COSP

Clear Other Speculative Prediction Restriction by Context prevents predictions, other than Cache prefetch, Control flow, and Data Value predictions, that predict execution addresses based on information gathered from earlier execution within a particular execution context. Predictions, other than Cache prefetch, Control flow, and Data Value predictions, determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control any speculative access occurring after the instruction is complete and synchronized.

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System  
(FEAT\_SPECRES2)

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

COSP RCTX, [<Xt>](#)

is equivalent to

[SYS](#) #3, C7, C3, #6, [<Xt>](#)

and is always the preferred disassembly.

Assembler Symbols

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

Operation

The description of [SYS](#) gives the operational pseudocode for this instruction.

CPP

Cache Prefetch Prediction Restriction by Context prevents cache allocation predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. The actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control cache prefetch predictions occurring after the instruction is complete and synchronized. For more information, see *CPP RCTX, Cache Prefetch Prediction Restriction by Context*.

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System  
(FEAT\_SPECRES)

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

CPP RCTX, [<Xt>](#)  
  
is equivalent to  
  
[SYS](#) #3, C7, C3, #7, [<Xt>](#)  
  
and is always the preferred disassembly.

Assembler Symbols

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

Operation

The description of [SYS](#) gives the operational pseudocode for this instruction.

## CPYFP, CPYFM, CPYFE

Memory Copy Forward-only. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFP, then CPYFM, and then CPYFE. CPYFP performs some preconditioning of the arguments suitable for using the CPYFM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFM performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFE performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFP, option A (which results in encoding PSTATE.C = 0):

- If  $X_n < 63$  == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1\* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFP, option B (which results in encoding PSTATE.C = 1):

- If  $X_n < 63$  == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFM, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1\* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1\* the number of bytes remaining to be copied in the memory copy in total.

For CPYFM, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFE, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1\* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFE, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.



- the value of Xd is written back with the lowest address that has not been copied to.

## Integer (FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																						
sz		0		1		1		0		0		1		op1		0		Rs						0				0				0				0				0		1		Rn						Rd			
																	op2																																				

### Epilogue (op1 == 10)

```
CPYFE [<Xd>]!, [<Xs>]!, <Xn>!
```

### Main (op1 == 01)

```
CPYFM [<Xd>]!, [<Xs>]!, <Xn>!
```

### Prologue (op1 == 00)

```
CPYFP [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSStage stage;
case op1 of
  when '00' stage = MOPSStage Prologue;
  when '01' stage = MOPSStage Main;
  when '10' stage = MOPSStage Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYFOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcvc = '0010';

else
    CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
        Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
        Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;

```

```
X[s, 64] = fromaddress;  
PSTATE.<N,Z,C,V> = nzcV;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYFPN, CPYFMN, CPYFEN

Memory Copy Forward-only, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPN, then CPYFMN, and then CPYFEN.

CPYFPN performs some preconditioning of the arguments suitable for using the CPYFMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPN, option A (which results in encoding PSTATE.C = 0):

- If  $X_n < 63$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF$ .
- $X_s$  holds the original  $X_s$  + saturated  $X_n$ .
- $X_d$  holds the original  $X_d$  + saturated  $X_n$ .
- $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPN, option B (which results in encoding PSTATE.C = 1):

- If  $X_n < 63$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF$ .
- $X_s$  holds the original  $X_s$  + an IMPLEMENTATION DEFINED number of bytes copied.
- $X_d$  holds the original  $X_d$  + an IMPLEMENTATION DEFINED number of bytes copied.
- $X_n$  holds the saturated  $X_n$  - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number and holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
- $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
- At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .

For CPYFMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
- $X_s$  holds the lowest address that the copy is copied from.
- $X_d$  holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of  $X_s$  is written back with the lowest address that has not been copied from.
  - the value of  $X_d$  is written back with the lowest address that has not been copied to.

For CPYFEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

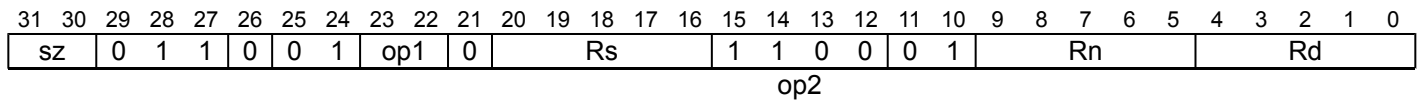
- $X_n$  is treated as a signed 64-bit number and holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
- $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
- At the end of the instruction, the value of  $X_n$  is written back with 0.

For CPYFEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
- $X_s$  holds the lowest address that the copy is copied from.
- $X_d$  holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of  $X_n$  is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

## Integer (FEAT\_MOPS)



### Epilogue (op1 == 10)

CPYFEN [<Xd>]!, [<Xs>]!, <Xn>!

### Main (op1 == 01)

CPYFMN [<Xd>]!, [<Xs>]!, <Xn>!

### Prologue (op1 == 00)

CPYFPN [<Xd>]!, [<Xs>]!, <Xn>!

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.





```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYFOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcvc = '0010';

else
    CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
        Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
        Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;

```

```
X[s, 64] = fromaddress;  
PSTATE.<N,Z,C,V> = nzcV;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYFPRN, CPYFMRN, CPYFERN

Memory Copy Forward-only, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRN, then CPYFMRN, and then CPYFERN.

CPYFPRN performs some preconditioning of the arguments suitable for using the CPYFMRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRN, option A (which results in encoding PSTATE.C = 0):

- If  $Xn < 63$  == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1\* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRN, option B (which results in encoding PSTATE.C = 1):

- If  $Xn < 63$  == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1\* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1\* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1\* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFERN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

## Integer (FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sz		0	1	1	0	0	1	op1		0	Rs					1	0	0	0	0	1	Rn					Rd				
																	op2														

### Epilogue (op1 == 10)

```
CPYFERN [<Xd>]!, [<Xs>]!, <Xn>!
```

### Main (op1 == 01)

```
CPYFMRN [<Xd>]!, [<Xs>]!, <Xn>!
```

### Prologue (op1 == 00)

```
CPYFPRN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYFOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcvc = '0010';

else
    CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
        Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
        Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;

```



```
X[s, 64] = fromaddress;  
PSTATE.<N,Z,C,V> = nzcV;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYFPRT, CPYFMRT, CPYFERT

Memory Copy Forward-only, reads unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRT, then CPYFMRT, and then CPYFERT.

CPYFPRT performs some preconditioning of the arguments suitable for using the CPYFMRT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERT performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRT, option A (which results in encoding PSTATE.C = 0):

- If  $Xn_{<63>} == 1$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF$ .
- $Xs$  holds the original  $Xs$  + saturated  $Xn$ .
- $Xd$  holds the original  $Xd$  + saturated  $Xn$ .
- $Xn$  holds  $-1 * \text{saturated } Xn + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .

After execution of CPYFPRT, option B (which results in encoding PSTATE.C = 1):

- If  $Xn_{<63>} == 1$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF$ .
- $Xs$  holds the original  $Xs$  + an IMPLEMENTATION DEFINED number of bytes copied.
- $Xd$  holds the original  $Xd$  + an IMPLEMENTATION DEFINED number of bytes copied.
- $Xn$  holds the saturated  $Xn$  - an IMPLEMENTATION DEFINED number of bytes copied.
- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .

For CPYFMRT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $Xn$  is treated as a signed 64-bit number and holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- $Xs$  holds the lowest address that the copy is copied from  $-Xn$ .
- $Xd$  holds the lowest address that the copy is made to  $-Xn$ .
- At the end of the instruction, the value of  $Xn$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .

For CPYFMRT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $Xn$  holds the number of bytes remaining to be copied in the memory copy in total.
- $Xs$  holds the lowest address that the copy is copied from.
- $Xd$  holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of  $Xn$  is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of  $Xs$  is written back with the lowest address that has not been copied from.
  - the value of  $Xd$  is written back with the lowest address that has not been copied to.

For CPYFERT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

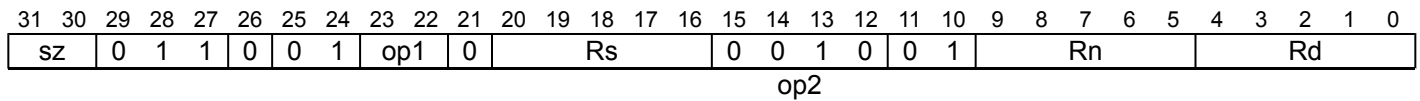
- $Xn$  is treated as a signed 64-bit number and holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- $Xs$  holds the lowest address that the copy is copied from  $-Xn$ .
- $Xd$  holds the lowest address that the copy is made to  $-Xn$ .
- At the end of the instruction, the value of  $Xn$  is written back with 0.

For CPYFERT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $Xn$  holds the number of bytes remaining to be copied in the memory copy in total.
- $Xs$  holds the lowest address that the copy is copied from.
- $Xd$  holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of  $Xn$  is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

## Integer (FEAT\_MOPS)



### Epilogue (op1 == 10)

```
CPYFERT [<Xd>]!, [<Xs>]!, <Xn>!
```

### Main (op1 == 01)

```
CPYFMRT [<Xd>]!, [<Xs>]!, <Xn>!
```

### Prologue (op1 == 00)

```
CPYFPRT [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYFOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcvc = '0010';

else
    CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
        Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
        Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;

```

```
X[s, 64] = fromaddress;  
PSTATE.<N,Z,C,V> = nzcV;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYFPRTN, CPYFMRTN, CPYFERTN

Memory Copy Forward-only, reads unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTN, then CPYFMRTN, and then CPYFERTN.

CPYFPRTN performs some preconditioning of the arguments suitable for using the CPYFMRTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERTN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRTN, option A (which results in encoding PSTATE.C = 0):

- If  $X_n < 63$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF$ .
- $X_s$  holds the original  $X_s$  + saturated  $X_n$ .
- $X_d$  holds the original  $X_d$  + saturated  $X_n$ .
- $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRTN, option B (which results in encoding PSTATE.C = 1):

- If  $X_n < 63$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF$ .
- $X_s$  holds the original  $X_s$  + an IMPLEMENTATION DEFINED number of bytes copied.
- $X_d$  holds the original  $X_d$  + an IMPLEMENTATION DEFINED number of bytes copied.
- $X_n$  holds the saturated  $X_n$  - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number and holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
- $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
- At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .

For CPYFMRTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
- $X_s$  holds the lowest address that the copy is copied from.
- $X_d$  holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of  $X_s$  is written back with the lowest address that has not been copied from.
  - the value of  $X_d$  is written back with the lowest address that has not been copied to.

For CPYFERTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number and holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
- $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
- At the end of the instruction, the value of  $X_n$  is written back with 0.

For CPYFERTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
- $X_s$  holds the lowest address that the copy is copied from.
- $X_d$  holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of  $X_n$  is written back with 0.



- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

## Integer (FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sz		0	1	1	0	0	1	op1		0	Rs					1	1	1	0	0	1	Rn					Rd				
op2																															

### Epilogue (op1 == 10)

```
CPYFERTN [<Xd>]!, [<Xs>]!, <Xn>!
```

### Main (op1 == 01)

```
CPYFMRTN [<Xd>]!, [<Xs>]!, <Xn>!
```

### Prologue (op1 == 00)

```
CPYFPRTN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYFOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcvc = '0010';

else
    CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
        Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
        Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;

```

```
X[s, 64] = fromaddress;  
PSTATE.<N,Z,C,V> = nzcV;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYFPRTRN, CPYFMRTRN, CPYFERTRN

Memory Copy Forward-only, reads unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTRN, then CPYFMRTRN, and then CPYFERTRN.

CPYFPRTRN performs some preconditioning of the arguments suitable for using the CPYFMRTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERTRN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRTRN, option A (which results in encoding PSTATE.C = 0):

- If  $Xn < 63$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF$ .
- $Xs$  holds the original  $Xs$  + saturated  $Xn$ .
- $Xd$  holds the original  $Xd$  + saturated  $Xn$ .
- $Xn$  holds  $-1 * \text{saturated } Xn + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRTRN, option B (which results in encoding PSTATE.C = 1):

- If  $Xn < 63$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF$ .
- $Xs$  holds the original  $Xs$  + an IMPLEMENTATION DEFINED number of bytes copied.
- $Xd$  holds the original  $Xd$  + an IMPLEMENTATION DEFINED number of bytes copied.
- $Xn$  holds the saturated  $Xn$  - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $Xn$  is treated as a signed 64-bit number and holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- $Xs$  holds the lowest address that the copy is copied from  $-Xn$ .
- $Xd$  holds the lowest address that the copy is made to  $-Xn$ .
- At the end of the instruction, the value of  $Xn$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .

For CPYFMRTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $Xn$  holds the number of bytes remaining to be copied in the memory copy in total.
- $Xs$  holds the lowest address that the copy is copied from.
- $Xd$  holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of  $Xn$  is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of  $Xs$  is written back with the lowest address that has not been copied from.
  - the value of  $Xd$  is written back with the lowest address that has not been copied to.

For CPYFERTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

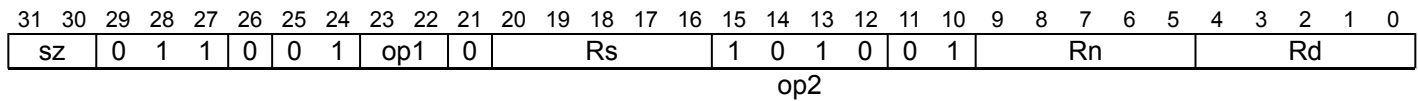
- $Xn$  is treated as a signed 64-bit number and holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- $Xs$  holds the lowest address that the copy is copied from  $-Xn$ .
- $Xd$  holds the lowest address that the copy is made to  $-Xn$ .
- At the end of the instruction, the value of  $Xn$  is written back with 0.

For CPYFERTRN option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $Xn$  holds the number of bytes remaining to be copied in the memory copy in total.
- $Xs$  holds the lowest address that the copy is copied from.
- $Xd$  holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of  $Xn$  is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

## Integer (FEAT\_MOPS)



### Epilogue (op1 == 10)

```
CPYFERTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

### Main (op1 == 01)

```
CPYFMRTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

### Prologue (op1 == 00)

```
CPYFPRTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.





```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYFOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcvc = '0010';

else
    CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
        Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
        Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;

```

```
X[s, 64] = fromaddress;  
PSTATE.<N,Z,C,V> = nzcV;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYFPRTWN, CPYFMRTWN, CPYFERTWN

Memory Copy Forward-only, reads unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTWN, then CPYFMRTWN, and then CPYFERTWN.

CPYFPRTWN performs some preconditioning of the arguments suitable for using the CPYFMRTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERTWN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRTWN, option A (which results in encoding PSTATE.C = 0):

- If  $Xn < 63$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF$ .
- $Xs$  holds the original  $Xs$  + saturated  $Xn$ .
- $Xd$  holds the original  $Xd$  + saturated  $Xn$ .
- $Xn$  holds  $-1 * \text{saturated } Xn + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .

After execution of CPYFPRTWN, option B (which results in encoding PSTATE.C = 1):

- If  $Xn < 63$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF$ .
- $Xs$  holds the original  $Xs$  + an IMPLEMENTATION DEFINED number of bytes copied.
- $Xd$  holds the original  $Xd$  + an IMPLEMENTATION DEFINED number of bytes copied.
- $Xn$  holds the saturated  $Xn$  - an IMPLEMENTATION DEFINED number of bytes copied.
- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .

For CPYFMRTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $Xn$  is treated as a signed 64-bit number and holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- $Xs$  holds the lowest address that the copy is copied from  $-Xn$ .
- $Xd$  holds the lowest address that the copy is made to  $-Xn$ .
- At the end of the instruction, the value of  $Xn$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .

For CPYFMRTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $Xn$  holds the number of bytes remaining to be copied in the memory copy in total.
- $Xs$  holds the lowest address that the copy is copied from.
- $Xd$  holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of  $Xn$  is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of  $Xs$  is written back with the lowest address that has not been copied from.
  - the value of  $Xd$  is written back with the lowest address that has not been copied to.

For CPYFERTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

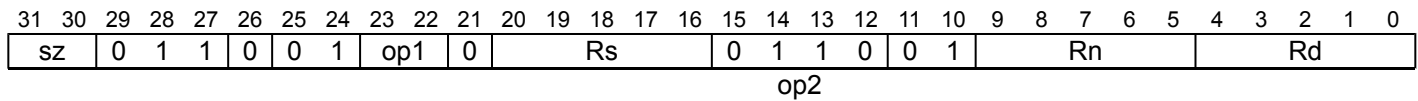
- $Xn$  is treated as a signed 64-bit number and holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- $Xs$  holds the lowest address that the copy is copied from  $-Xn$ .
- $Xd$  holds the lowest address that the copy is made to  $-Xn$ .
- At the end of the instruction, the value of  $Xn$  is written back with 0.

For CPYFERTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $Xn$  holds the number of bytes remaining to be copied in the memory copy in total.
- $Xs$  holds the lowest address that the copy is copied from.
- $Xd$  holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of  $Xn$  is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

## Integer (FEAT\_MOPS)



### Epilogue (op1 == 10)

CPYFERTWN [<Xd>]!, [<Xs>]!, <Xn>!

### Main (op1 == 01)

CPYFMRTWN [<Xd>]!, [<Xs>]!, <Xn>!

### Prologue (op1 == 00)

CPYFPRTWN [<Xd>]!, [<Xs>]!, <Xn>!

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYFOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcvc = '0010';

else
    CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
        Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
        Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;

```



```
X[s, 64] = fromaddress;  
PSTATE.<N,Z,C,V> = nzcV;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYFPT, CPYFMT, CPYFET

Memory Copy Forward-only, reads and writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPT, then CPYFMT, and then CPYFET.

CPYFPT performs some preconditioning of the arguments suitable for using the CPYFMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFET performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPT, option A (which results in encoding PSTATE.C = 0):

- If  $Xn < 63$  == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1\* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPT, option B (which results in encoding PSTATE.C = 1):

- If  $Xn < 63$  == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1\* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1\* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFET, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1\* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFET, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

## Integer (FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sz		0	1	1	0	0	1	op1		0	Rs					0	0	1	1	0	1	Rn					Rd				
op2																															

### Epilogue (op1 == 10)

CPYFET [<Xd>]!, [<Xs>]!, <Xn>!

### Main (op1 == 01)

CPYFMT [<Xd>]!, [<Xs>]!, <Xn>!

### Prologue (op1 == 00)

CPYFPT [<Xd>]!, [<Xs>]!, <Xn>!

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYFOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rntontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcvc = '0010';

else
    CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
        Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
        Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;

```

```
X[s, 64] = fromaddress;  
PSTATE.<N,Z,C,V> = nzcV;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYFPTN, CPYFMTN, CPYFETN

Memory Copy Forward-only, reads and writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTN, then CPYFMTN, and then CPYFETN.

CPYFPTN performs some preconditioning of the arguments suitable for using the CPYFMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFETN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPTN, option A (which results in encoding PSTATE.C = 0):

- If  $X_n < 63$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF$ .
- $X_s$  holds the original  $X_s$  + saturated  $X_n$ .
- $X_d$  holds the original  $X_d$  + saturated  $X_n$ .
- $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPTN, option B (which results in encoding PSTATE.C = 1):

- If  $X_n < 63$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF$ .
- $X_s$  holds the original  $X_s$  + an IMPLEMENTATION DEFINED number of bytes copied.
- $X_d$  holds the original  $X_d$  + an IMPLEMENTATION DEFINED number of bytes copied.
- $X_n$  holds the saturated  $X_n$  - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number and holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
- $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
- At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .

For CPYFMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
- $X_s$  holds the lowest address that the copy is copied from.
- $X_d$  holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of  $X_s$  is written back with the lowest address that has not been copied from.
  - the value of  $X_d$  is written back with the lowest address that has not been copied to.

For CPYFETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number and holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
- $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
- At the end of the instruction, the value of  $X_n$  is written back with 0.

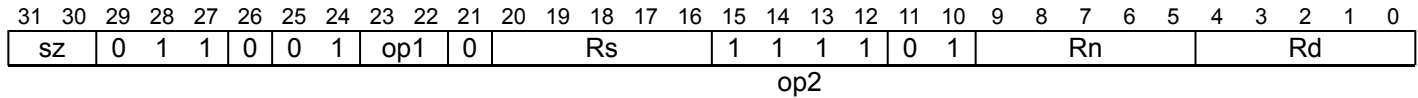
For CPYFETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
- $X_s$  holds the lowest address that the copy is copied from.
- $X_d$  holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of  $X_n$  is written back with 0.



- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

## Integer (FEAT\_MOPS)



### Epilogue (op1 == 10)

```
CPYFETN [<Xd>]!, [<Xs>]!, <Xn>!
```

### Main (op1 == 01)

```
CPYFMTN [<Xd>]!, [<Xs>]!, <Xn>!
```

### Prologue (op1 == 00)

```
CPYFPTN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYFOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcvc = '0010';

else
    CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
        Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
        Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;

```

```
X[s, 64] = fromaddress;  
PSTATE.<N,Z,C,V> = nzcV;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYFPTRN, CPYFMTRN, CPYFETRN

Memory Copy Forward-only, reads and writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTRN, then CPYFMTRN, and then CPYFETRN.

CPYFPTRN performs some preconditioning of the arguments suitable for using the CPYFMTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFETRN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPTRN, option A (which results in encoding PSTATE.C = 0):

- If  $X_n < 63 > == 1$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF$ .
- $X_s$  holds the original  $X_s$  + saturated  $X_n$ .
- $X_d$  holds the original  $X_d$  + saturated  $X_n$ .
- $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPTRN, option B (which results in encoding PSTATE.C = 1):

- If  $X_n < 63 > == 1$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF$ .
- $X_s$  holds the original  $X_s$  + an IMPLEMENTATION DEFINED number of bytes copied.
- $X_d$  holds the original  $X_d$  + an IMPLEMENTATION DEFINED number of bytes copied.
- $X_n$  holds the saturated  $X_n$  - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number and holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
- $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
- At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .

For CPYFMTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
- $X_s$  holds the lowest address that the copy is copied from.
- $X_d$  holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of  $X_s$  is written back with the lowest address that has not been copied from.
  - the value of  $X_d$  is written back with the lowest address that has not been copied to.

For CPYFETRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number and holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
- $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
- At the end of the instruction, the value of  $X_n$  is written back with 0.

For CPYFETRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
- $X_s$  holds the lowest address that the copy is copied from.
- $X_d$  holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of  $X_n$  is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

## Integer (FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sz		0	1	1	0	0	1	op1		0	Rs					1	0	1	1	0	1	Rn					Rd				
																	op2														

### Epilogue (op1 == 10)

```
CPYFETR [ <Xd> ]!, [ <Xs> ]!, <Xn>!
```

### Main (op1 == 01)

```
CPYFMTR [ <Xd> ]!, [ <Xs> ]!, <Xn>!
```

### Prologue (op1 == 00)

```
CPYFPTR [ <Xd> ]!, [ <Xs> ]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.





```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYFOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcvc = '0010';

else
    CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
        Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
        Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;

```

```
X[s, 64] = fromaddress;  
PSTATE.<N,Z,C,V> = nzcV;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYFPTWN, CPYFMTWN, CPYFETWN

Memory Copy Forward-only, reads and writes unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTWN, then CPYFMTWN, and then CPYFETWN.

CPYFPTWN performs some preconditioning of the arguments suitable for using the CPYFMTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFETWN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPTWN, option A (which results in encoding PSTATE.C = 0):

- If  $Xn < 63$  == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1\* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPTWN, option B (which results in encoding PSTATE.C = 1):

- If  $Xn < 63$  == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1\* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1\* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFETWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1\* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

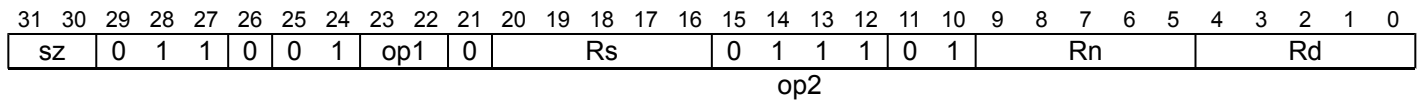
For CPYFETWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

## Integer

(FEAT\_MOPS)



### Epilogue (op1 == 10)

```
CPYFETWN [<Xd>]!, [<Xs>]!, <Xn>!
```

### Main (op1 == 01)

```
CPYFMTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

### Prologue (op1 == 00)

```
CPYFPTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYFOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcvc = '0010';

else
    CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
        Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
        Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;

```



```
X[s, 64] = fromaddress;  
PSTATE.<N,Z,C,V> = nzcV;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYFPWN, CPYFMWN, CPYFEWN

Memory Copy Forward-only, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWN, then CPYFMWN, and then CPYFEWN.

CPYFPWN performs some preconditioning of the arguments suitable for using the CPYFMWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWN, option A (which results in encoding PSTATE.C = 0):

- If  $Xn < 63$  == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1\* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWN, option B (which results in encoding PSTATE.C = 1):

- If  $Xn < 63$  == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1\* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1\* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1\* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

## Integer (FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0														
sz		0		1		1		0		0		1		op1		0		Rs						0		1		0		0		0		1		Rn						Rd			
																	op2																												

### Epilogue (op1 == 10)

```
CPYFEWN [<Xd>]!, [<Xs>]!, <Xn>!
```

### Main (op1 == 01)

```
CPYFMWN [<Xd>]!, [<Xs>]!, <Xn>!
```

### Prologue (op1 == 00)

```
CPYFPWN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYFOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcvc = '0010';

else
    CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
        Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
        Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;

```

```
X[s, 64] = fromaddress;  
PSTATE.<N,Z,C,V> = nzcV;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYFPWT, CPYFMWT, CPYFEWT

Memory Copy Forward-only, writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWT, then CPYFMWT, and then CPYFEWT.

CPYFPWT performs some preconditioning of the arguments suitable for using the CPYFMWT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWT performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWT, option A (which results in encoding PSTATE.C = 0):

- If  $Xn_{<63>} == 1$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF$ .
- $Xs$  holds the original  $Xs$  + saturated  $Xn$ .
- $Xd$  holds the original  $Xd$  + saturated  $Xn$ .
- $Xn$  holds  $-1 * \text{saturated } Xn + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .

After execution of CPYFPWT, option B (which results in encoding PSTATE.C = 1):

- If  $Xn_{<63>} == 1$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF$ .
- $Xs$  holds the original  $Xs$  + an IMPLEMENTATION DEFINED number of bytes copied.
- $Xd$  holds the original  $Xd$  + an IMPLEMENTATION DEFINED number of bytes copied.
- $Xn$  holds the saturated  $Xn$  - an IMPLEMENTATION DEFINED number of bytes copied.
- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .

For CPYFMWT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $Xn$  is treated as a signed 64-bit number and holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- $Xs$  holds the lowest address that the copy is copied from  $-Xn$ .
- $Xd$  holds the lowest address that the copy is made to  $-Xn$ .
- At the end of the instruction, the value of  $Xn$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .

For CPYFMWT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $Xn$  holds the number of bytes remaining to be copied in the memory copy in total.
- $Xs$  holds the lowest address that the copy is copied from.
- $Xd$  holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of  $Xn$  is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of  $Xs$  is written back with the lowest address that has not been copied from.
  - the value of  $Xd$  is written back with the lowest address that has not been copied to.

For CPYFEWT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $Xn$  is treated as a signed 64-bit number and holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- $Xs$  holds the lowest address that the copy is copied from  $-Xn$ .
- $Xd$  holds the lowest address that the copy is made to  $-Xn$ .
- At the end of the instruction, the value of  $Xn$  is written back with 0.

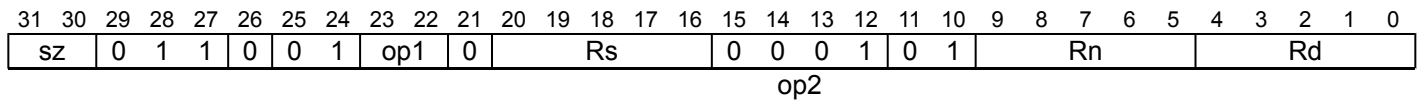
For CPYFEWT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $Xn$  holds the number of bytes remaining to be copied in the memory copy in total.
- $Xs$  holds the lowest address that the copy is copied from.
- $Xd$  holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of  $Xn$  is written back with 0.



- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

## Integer (FEAT\_MOPS)



### Epilogue (op1 == 10)

```
CPYFEWT [<Xd>]!, [<Xs>]!, <Xn>!
```

### Main (op1 == 01)

```
CPYFMWT [<Xd>]!, [<Xs>]!, <Xn>!
```

### Prologue (op1 == 00)

```
CPYFPWT [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYFOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rntontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcvc = '0010';

else
    CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
        Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
        Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;

```

```
X[s, 64] = fromaddress;  
PSTATE.<N,Z,C,V> = nzcV;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYFPWTN, CPYFMWTN, CPYFEWTN

Memory Copy Forward-only, writes unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTN, then CPYFMWTN, and then CPYFEWTN.

CPYFPWTN performs some preconditioning of the arguments suitable for using the CPYFMWTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWTN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWTN, option A (which results in encoding PSTATE.C = 0):

- If  $Xn < 63$  == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1\* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWTN, option B (which results in encoding PSTATE.C = 1):

- If  $Xn < 63$  == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1\* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1\* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1\* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

## Integer (FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0														
sz		0		1		1		0		0		1		op1		0		Rs						1		1		0		1		0		1		Rn						Rd			
																	op2																												

### Epilogue (op1 == 10)

```
CPYFEWTN [<Xd>]!, [<Xs>]!, <Xn>!
```

### Main (op1 == 01)

```
CPYFMWTN [<Xd>]!, [<Xs>]!, <Xn>!
```

### Prologue (op1 == 00)

```
CPYFPWTN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.





```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYFOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcvc = '0010';

else
    CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
        Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
        Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;

```

```
X[s, 64] = fromaddress;  
PSTATE.<N,Z,C,V> = nzcV;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYFPWTRN, CPYFMWTRN, CPYFEWTRN

Memory Copy Forward-only, writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTRN, then CPYFMWTRN, and then CPYFEWTRN.

CPYFPWTRN performs some preconditioning of the arguments suitable for using the CPYFMWTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWTRN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWTRN, option A (which results in encoding PSTATE.C = 0):

- If  $Xn < 63$  == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1\* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWTRN, option B (which results in encoding PSTATE.C = 1):

- If  $Xn < 63$  == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1\* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1\* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

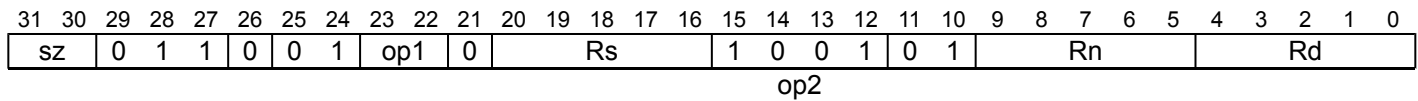
- Xn is treated as a signed 64-bit number and holds -1\* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

## Integer (FEAT\_MOPS)



### Epilogue (op1 == 10)

```
CPYFEWTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

### Main (op1 == 01)

```
CPYFMWTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

### Prologue (op1 == 00)

```
CPYFPWTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYFOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcvc = '0010';

else
    CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
        Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
        Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;

```



```
X[s, 64] = fromaddress;  
PSTATE.<N,Z,C,V> = nzcV;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYFPWTWN, CPYFMWTWN, CPYFEWTWN

Memory Copy Forward-only, writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTWN, then CPYFMWTWN, and then CPYFEWTWN.

CPYFPWTWN performs some preconditioning of the arguments suitable for using the CPYFMWTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWTWN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWTWN, option A (which results in encoding PSTATE.C = 0):

- If  $Xn < 63$  == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1\* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWTWN, option B (which results in encoding PSTATE.C = 1):

- If  $Xn < 63$  == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1\* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1\* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

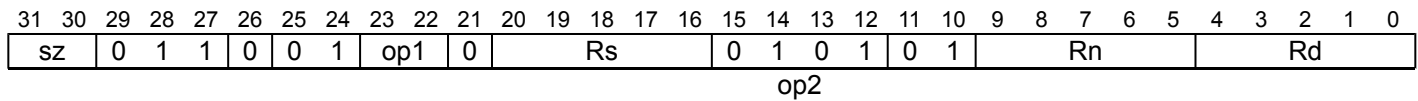
- Xn is treated as a signed 64-bit number and holds -1\* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

## Integer (FEAT\_MOPS)



### Epilogue (op1 == 10)

```
CPYFEWTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

### Main (op1 == 01)

```
CPYFMWTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

### Prologue (op1 == 00)

```
CPYFPWTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYFOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcvc = '0010';

else
    CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
        Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
        Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;

```

```
X[s, 64] = fromaddress;  
PSTATE.<N,Z,C,V> = nzcV;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYP, CPYM, CPYE

Memory Copy. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYP, then CPYM, and then CPYE.

CPYP performs some preconditioning of the arguments suitable for using the CPYM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYM performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYE performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYP, the following saturation logic is applied:

If  $X_n < 63:55 \neq 000000000$ , the copy size  $X_n$  is saturated to  $0x007FFFFFFFFFFFFFFF$ .

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If  $(X_s > X_d) \&\& (X_d + \text{saturated } X_n) > X_s$ , then direction = forward

Elsif  $(X_s < X_d) \&\& (X_s + \text{saturated } X_n) > X_d$ , then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYP, option A (which results in encoding  $PSTATE.C = 0$ ):

- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n$ .
  - $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- If the copy is in the backward direction, then:
  - $X_s$  and  $X_d$  are unchanged.
  - $X_n$  holds the saturated value of  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .

After execution of CPYP, option B (which results in encoding  $PSTATE.C = 1$ ):

- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the backward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{1,0,0\}$ .

For CPYM, option A (encoded by  $PSTATE.C = 0$ ), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number.
- If the copy is in the forward direction ( $X_n$  is a negative number), then:
  - $X_n$  holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
  - $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
  - $X_d$  holds the lowest address that the copy is copied to  $-X_n$ .
  - At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- If the copy is in the backward direction ( $X_n$  is a positive number), then:
  - $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$  holds the highest address that the copy is copied from  $-X_n+1$ .
  - $X_d$  holds the highest address that the copy is copied to  $-X_n+1$ .
  - At the end of the instruction, the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYM, option B (encoded by  $PSTATE.C = 1$ ), the format of the arguments is:

- $X_n$  holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction ( $PSTATE.N == 0$ ), then:



- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYE, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1\* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is made to.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYE, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from.
  - Xd holds the highest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

## Integer

(FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																		
sz		0		1		1		1		0		1		op1		0		Rs				0				0				0				0				0		1		Rn				Rd			
																op2																																	

## Epilogue (op1 == 10)

```
CPYE [<Xd>]!, [<Xs>]!, <Xn>!
```

## Main (op1 == 01)

```
CPYM [<Xd>]!, [<Xs>]!, <Xn>!
```

## Prologue (op1 == 00)

```
CPYP [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd>** For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs>** For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn>** For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFFFF<63:0>;

    boolean forward = IsMemCpyForward(toaddress, fromaddress, cpysize);

    if implements_option_a then
        nzcvc = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                nzcvc = '1010';
            else
                nzcvc = '0010';
        else
            CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            if stage != MOPSSStage\_Prologue then
                X[n, 64] = cpysize;
        else
            while UInt(stagecpysize) > 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= UInt(stagecpysize);

```

```

if nzcv<3> == '0' then // PSTATE.N
    readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
    Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
else
    readdata<B*8-1:0> = Mem[fromaddress-B, B, raccdesc];
    Mem[toaddress-B, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress - B;
    toaddress = toaddress - B;

cpysize = cpysize - B;
stagecpysize = stagecpysize - B;

if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYPN, CPYMN, CPYEN

Memory Copy, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPN, then CPYMN, and then CPYEN.

CPYPN performs some preconditioning of the arguments suitable for using the CPYMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPN, the following saturation logic is applied:

If  $X_n < 63:55 > \neq 000000000$ , the copy size  $X_n$  is saturated to  $0x007FFFFFFFFFFFFFFF$ .

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If  $(X_s > X_d) \&\& (X_d + \text{saturated } X_n) > X_s$ , then direction = forward

Elsif  $(X_s < X_d) \&\& (X_s + \text{saturated } X_n) > X_d$ , then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPN, option A (which results in encoding  $PSTATE.C = 0$ ):

- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n$ .
  - $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- If the copy is in the backward direction, then:
  - $X_s$  and  $X_d$  are unchanged.
  - $X_n$  holds the saturated value of  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .

After execution of CPYPN, option B (which results in encoding  $PSTATE.C = 1$ ):

- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the backward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{1,0,0\}$ .

For CPYMN, option A (encoded by  $PSTATE.C = 0$ ), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number.
- If the copy is in the forward direction ( $X_n$  is a negative number), then:
  - $X_n$  holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
  - $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
  - $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
  - At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- If the copy is in the backward direction ( $X_n$  is a positive number), then:
  - $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$  holds the highest address that the copy is copied from  $-X_n+1$ .
  - $X_d$  holds the highest address that the copy is copied to  $-X_n+1$ .
  - At the end of the instruction, the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMN, option B (encoded by  $PSTATE.C = 1$ ), the format of the arguments is:

- $X_n$  holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1\* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

## Integer

(FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sz		0		1		1		1		0		1		op1		0		Rs				1		1		0		0		0		1		Rn				Rd			
																op2																									

## Epilogue (op1 == 10)

```
CPYEN [<Xd>]!, [<Xs>]!, <Xn>!
```

## Main (op1 == 01)

```
CPYMN [<Xd>]!, [<Xs>]!, <Xn>!
```

## Prologue (op1 == 00)

```
CPYPN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.





```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFFFF<63:0>;

    boolean forward = IsMemCpyForward(toaddress, fromaddress, cpysize);

    if implements_option_a then
        nzcvc = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                nzcvc = '1010';
            else
                nzcvc = '0010';
        else
            CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            if stage != MOPSSStage\_Prologue then
                X[n, 64] = cpysize;
        else
            while UInt(stagecpysize) > 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= UInt(stagecpysize);

```

```

if nzcv<3> == '0' then // PSTATE.N
    readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
    Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
else
    readdata<B*8-1:0> = Mem[fromaddress-B, B, raccdesc];
    Mem[toaddress-B, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress - B;
    toaddress = toaddress - B;

cpysize = cpysize - B;
stagecpysize = stagecpysize - B;

if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYPRN, CPYMRN, CPYERN

Memory Copy, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRN, then CPYMRN, and then CPYERN.

CPYPRN performs some preconditioning of the arguments suitable for using the CPYMRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRN, the following saturation logic is applied:

If  $X_n < 63:55 > \neq 000000000$ , the copy size  $X_n$  is saturated to  $0x007FFFFFFFFFFFFFFF$ .

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If  $(X_s > X_d) \&\& (X_d + \text{saturated } X_n) > X_s$ , then direction = forward

Elsif  $(X_s < X_d) \&\& (X_s + \text{saturated } X_n) > X_d$ , then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRN, option A (which results in encoding  $PSTATE.C = 0$ ):

- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n$ .
  - $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- If the copy is in the backward direction, then:
  - $X_s$  and  $X_d$  are unchanged.
  - $X_n$  holds the saturated value of  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .

After execution of CPYPRN, option B (which results in encoding  $PSTATE.C = 1$ ):

- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the backward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{1,0,0\}$ .

For CPYMRN, option A (encoded by  $PSTATE.C = 0$ ), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number.
- If the copy is in the forward direction ( $X_n$  is a negative number), then:
  - $X_n$  holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
  - $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
  - $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
  - At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- If the copy is in the backward direction ( $X_n$  is a positive number), then:
  - $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$  holds the highest address that the copy is copied from  $-X_n+1$ .
  - $X_d$  holds the highest address that the copy is copied to  $-X_n+1$ .
  - At the end of the instruction, the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRN, option B (encoded by  $PSTATE.C = 1$ ), the format of the arguments is:

- $X_n$  holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1\* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYERN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

## Integer

(FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																						
sz		0		1		1		1		0		1		op1		0		Rs				1				0				0				0				0				1				Rn				Rd			
																op2																																					

## Epilogue (op1 == 10)

```
CPYERN [<Xd>]!, [<Xs>]!, <Xn>!
```

## Main (op1 == 01)

```
CPYMRN [<Xd>]!, [<Xs>]!, <Xn>!
```

## Prologue (op1 == 00)

```
CPYPRN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFFFF<63:0>;

    boolean forward = IsMemCpyForward(toaddress, fromaddress, cpysize);

    if implements_option_a then
        nzcvc = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                nzcvc = '1010';
            else
                nzcvc = '0010';
        else
            CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            if stage != MOPSSStage\_Prologue then
                X[n, 64] = cpysize;
        else
            while UInt(stagecpysize) > 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= UInt(stagecpysize);

```



```

if nzcv<3> == '0' then // PSTATE.N
    readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
    Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
else
    readdata<B*8-1:0> = Mem[fromaddress-B, B, raccdesc];
    Mem[toaddress-B, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress - B;
    toaddress = toaddress - B;

cpysize = cpysize - B;
stagecpysize = stagecpysize - B;

if stage != MOPSSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;

if stage == MOPSSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYPRT, CPYMRT, CPYERT

Memory Copy, reads unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRT, then CPYMRT, and then CPYERT.

CPYPRT performs some preconditioning of the arguments suitable for using the CPYMRT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERT performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRT, the following saturation logic is applied:

If  $X_n < 63:55 > \neq 000000000$ , the copy size  $X_n$  is saturated to  $0x007FFFFFFFFFFFFFFF$ .

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If  $(X_s > X_d) \&\& (X_d + \text{saturated } X_n) > X_s$ , then direction = forward

Elsif  $(X_s < X_d) \&\& (X_s + \text{saturated } X_n) > X_d$ , then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRT, option A (which results in encoding  $PSTATE.C = 0$ ):

- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n$ .
  - $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- If the copy is in the backward direction, then:
  - $X_s$  and  $X_d$  are unchanged.
  - $X_n$  holds the saturated value of  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .

After execution of CPYPRT, option B (which results in encoding  $PSTATE.C = 1$ ):

- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the backward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{1,0,0\}$ .

For CPYMRT, option A (encoded by  $PSTATE.C = 0$ ), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number.
- If the copy is in the forward direction ( $X_n$  is a negative number), then:
  - $X_n$  holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
  - $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
  - $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
  - At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- If the copy is in the backward direction ( $X_n$  is a positive number), then:
  - $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$  holds the highest address that the copy is copied from  $-X_n+1$ .
  - $X_d$  holds the highest address that the copy is copied to  $-X_n+1$ .
  - At the end of the instruction, the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRT, option B (encoded by  $PSTATE.C = 1$ ), the format of the arguments is:

- $X_n$  holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1\* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYERT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

## Integer

(FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sz		0		1		1		1		0		1		op1		0		Rs				0		0		1		0		0		1		Rn				Rd			
																op2																									

## Epilogue (op1 == 10)

```
CPYERT [<Xd>]!, [<Xs>]!, <Xn>!
```

## Main (op1 == 01)

```
CPYMRT [<Xd>]!, [<Xs>]!, <Xn>!
```

## Prologue (op1 == 00)

```
CPYPRT [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rntontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFFFF<63:0>;

    boolean forward = IsMemCpyForward(toaddress, fromaddress, cpysize);

    if implements_option_a then
        nzcvc = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                nzcvc = '1010';
            else
                nzcvc = '0010';
    else
        CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            if stage != MOPSSStage\_Prologue then
                X[n, 64] = cpysize;
    else
        while UInt(stagecpysize) > 0 do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(toaddress, fromaddress, cpysize);
            assert B <= UInt(stagecpysize);

```

```

if nzcv<3> == '0' then // PSTATE.N
    readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
    Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
else
    readdata<B*8-1:0> = Mem[fromaddress-B, B, raccdesc];
    Mem[toaddress-B, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress - B;
    toaddress = toaddress - B;

cpysize = cpysize - B;
stagecpysize = stagecpysize - B;

if stage != MOPSSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;

if stage == MOPSSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYPRTN, CPYMRTN, CPYERTN

Memory Copy, reads unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTN, then CPYMRTN, and then CPYERTN.

CPYPRTN performs some preconditioning of the arguments suitable for using the CPYMRTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERTN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRTN, the following saturation logic is applied:

If  $X_n < 63:55 > \neq 000000000$ , the copy size  $X_n$  is saturated to  $0x007FFFFFFFFFFFFFFF$ .

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If  $(X_s > X_d) \&\& (X_d + \text{saturated } X_n) > X_s$ , then direction = forward

Elsif  $(X_s < X_d) \&\& (X_s + \text{saturated } X_n) > X_d$ , then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRTN, option A (which results in encoding  $PSTATE.C = 0$ ):

- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n$ .
  - $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- If the copy is in the backward direction, then:
  - $X_s$  and  $X_d$  are unchanged.
  - $X_n$  holds the saturated value of  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .

After execution of CPYPRTN, option B (which results in encoding  $PSTATE.C = 1$ ):

- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the backward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{1,0,0\}$ .

For CPYMRTN, option A (encoded by  $PSTATE.C = 0$ ), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number.
- If the copy is in the forward direction ( $X_n$  is a negative number), then:
  - $X_n$  holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
  - $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
  - $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
  - At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- If the copy is in the backward direction ( $X_n$  is a positive number), then:
  - $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$  holds the highest address that the copy is copied from  $-X_n+1$ .
  - $X_d$  holds the highest address that the copy is copied to  $-X_n+1$ .
  - At the end of the instruction, the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRTN, option B (encoded by  $PSTATE.C = 1$ ), the format of the arguments is:

- $X_n$  holds the number of bytes to be copied in the memory copy in total.



- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1\* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYERTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

## Integer

(FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sz		0		1		1		1		0		1		op1		0		Rs				1		1		1		0		0		1		Rn				Rd			
																op2																									

## Epilogue (op1 == 10)

```
CPYERTN [<Xd>]!, [<Xs>]!, <Xn>!
```

## Main (op1 == 01)

```
CPYMRTN [<Xd>]!, [<Xs>]!, <Xn>!
```

## Prologue (op1 == 00)

```
CPYPRTN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFFFF<63:0>;

    boolean forward = IsMemCpyForward(toaddress, fromaddress, cpysize);

    if implements_option_a then
        nzcvc = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                nzcvc = '1010';
            else
                nzcvc = '0010';
    else
        CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            if stage != MOPSSStage\_Prologue then
                X[n, 64] = cpysize;
    else
        while UInt(stagecpysize) > 0 do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(toaddress, fromaddress, cpysize);
            assert B <= UInt(stagecpysize);

```

```

if nzcv<3> == '0' then // PSTATE.N
    readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
    Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
else
    readdata<B*8-1:0> = Mem[fromaddress-B, B, raccdesc];
    Mem[toaddress-B, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress - B;
    toaddress = toaddress - B;

cpysize = cpysize - B;
stagecpysize = stagecpysize - B;

if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYPRTRN, CPYMRTRN, CPYERTRN

Memory Copy, reads unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTRN, then CPYMRTRN, and then CPYERTRN.

CPYPRTRN performs some preconditioning of the arguments suitable for using the CPYMRTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERTRN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRTRN, the following saturation logic is applied:

If  $X_n < 63:55 > \neq 000000000$ , the copy size  $X_n$  is saturated to  $0x007FFFFFFFFFFFFFFF$ .

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If  $(X_s > X_d) \&\& (X_d + \text{saturated } X_n) > X_s$ , then direction = forward

Elsif  $(X_s < X_d) \&\& (X_s + \text{saturated } X_n) > X_d$ , then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRTRN, option A (which results in encoding  $PSTATE.C = 0$ ):

- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n$ .
  - $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- If the copy is in the backward direction, then:
  - $X_s$  and  $X_d$  are unchanged.
  - $X_n$  holds the saturated value of  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .

After execution of CPYPRTRN, option B (which results in encoding  $PSTATE.C = 1$ ):

- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the backward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{1,0,0\}$ .

For CPYMRTRN, option A (encoded by  $PSTATE.C = 0$ ), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number.
- If the copy is in the forward direction ( $X_n$  is a negative number), then:
  - $X_n$  holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
  - $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
  - $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
  - At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- If the copy is in the backward direction ( $X_n$  is a positive number), then:
  - $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$  holds the highest address that the copy is copied from  $-X_n+1$ .
  - $X_d$  holds the highest address that the copy is copied to  $-X_n+1$ .
  - At the end of the instruction, the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRTRN, option B (encoded by  $PSTATE.C = 1$ ), the format of the arguments is:

- $X_n$  holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1\* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYERTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

## Integer

(FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sz		0		1		1		1		0		1		op1		0		Rs				1		0		1		0		0		1		Rn				Rd			
op2																																									

## Epilogue (op1 == 10)

CPYERTRN [<Xd>]!, [<Xs>]!, <Xn>!

## Main (op1 == 01)

CPYMRTRN [<Xd>]!, [<Xs>]!, <Xn>!

## Prologue (op1 == 00)

CPYPRTRN [<Xd>]!, [<Xs>]!, <Xn>!

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.





```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFFFF<63:0>;

    boolean forward = IsMemCpyForward(toaddress, fromaddress, cpysize);

    if implements_option_a then
        nzcvc = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                nzcvc = '1010';
            else
                nzcvc = '0010';
    else
        CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            if stage != MOPSSStage\_Prologue then
                X[n, 64] = cpysize;
    else
        while UInt(stagecpysize) > 0 do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(toaddress, fromaddress, cpysize);
            assert B <= UInt(stagecpysize);

```

```

if nzcv<3> == '0' then // PSTATE.N
    readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
    Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
else
    readdata<B*8-1:0> = Mem[fromaddress-B, B, raccdesc];
    Mem[toaddress-B, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress - B;
    toaddress = toaddress - B;

cpysize = cpysize - B;
stagecpysize = stagecpysize - B;

if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYPRTWN, CPYMRTWN, CPYERTWN

Memory Copy, reads unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTWN, then CPYMRTWN, and then CPYERTWN.

CPYPRTWN performs some preconditioning of the arguments suitable for using the CPYMRTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERTWN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRTWN, the following saturation logic is applied:

If  $X_n < 63:55 > \neq 000000000$ , the copy size  $X_n$  is saturated to  $0x007FFFFFFFFFFFFFFF$ .

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If  $(X_s > X_d) \&\& (X_d + \text{saturated } X_n) > X_s$ , then direction = forward

Elsif  $(X_s < X_d) \&\& (X_s + \text{saturated } X_n) > X_d$ , then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRTWN, option A (which results in encoding  $PSTATE.C = 0$ ):

- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n$ .
  - $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- If the copy is in the backward direction, then:
  - $X_s$  and  $X_d$  are unchanged.
  - $X_n$  holds the saturated value of  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .

After execution of CPYPRTWN, option B (which results in encoding  $PSTATE.C = 1$ ):

- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the backward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{1,0,0\}$ .

For CPYMRTWN, option A (encoded by  $PSTATE.C = 0$ ), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number.
- If the copy is in the forward direction ( $X_n$  is a negative number), then:
  - $X_n$  holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
  - $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
  - $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
  - At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- If the copy is in the backward direction ( $X_n$  is a positive number), then:
  - $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$  holds the highest address that the copy is copied from  $-X_n+1$ .
  - $X_d$  holds the highest address that the copy is copied to  $-X_n+1$ .
  - At the end of the instruction, the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRTWN, option B (encoded by  $PSTATE.C = 1$ ), the format of the arguments is:

- $X_n$  holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1\* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYERTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

## Integer

(FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sz		0		1		1		1		0		1		op1		0		Rs				0		1		1		0		0		1		Rn				Rd			
																op2																									

## Epilogue (op1 == 10)

CPYERTWN [<Xd>]!, [<Xs>]!, <Xn>!

## Main (op1 == 01)

CPYMRTWN [<Xd>]!, [<Xs>]!, <Xn>!

## Prologue (op1 == 00)

CPYPRTWN [<Xd>]!, [<Xs>]!, <Xn>!

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFFFF<63:0>;

    boolean forward = IsMemCpyForward(toaddress, fromaddress, cpysize);

    if implements_option_a then
        nzcvc = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                nzcvc = '1010';
            else
                nzcvc = '0010';
        else
            CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            if stage != MOPSSStage\_Prologue then
                X[n, 64] = cpysize;
        else
            while UInt(stagecpysize) > 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= UInt(stagecpysize);

```



```

if nzcv<3> == '0' then // PSTATE.N
    readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
    Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
else
    readdata<B*8-1:0> = Mem[fromaddress-B, B, raccdesc];
    Mem[toaddress-B, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress - B;
    toaddress = toaddress - B;

cpysize = cpysize - B;
stagecpysize = stagecpysize - B;

if stage != MOPSSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;

if stage == MOPSSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYPT, CPYMT, CPYET

Memory Copy, reads and writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPT, then CPYMT, and then CPYET.

CPYPT performs some preconditioning of the arguments suitable for using the CPYMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYET performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPT, the following saturation logic is applied:

If  $X_n < 63:55 > \neq 000000000$ , the copy size  $X_n$  is saturated to  $0x007FFFFFFFFFFFFFFF$ .

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If  $(X_s > X_d) \&\& (X_d + \text{saturated } X_n) > X_s$ , then direction = forward

Elsif  $(X_s < X_d) \&\& (X_s + \text{saturated } X_n) > X_d$ , then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPT, option A (which results in encoding  $PSTATE.C = 0$ ):

- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n$ .
  - $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- If the copy is in the backward direction, then:
  - $X_s$  and  $X_d$  are unchanged.
  - $X_n$  holds the saturated value of  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .

After execution of CPYPT, option B (which results in encoding  $PSTATE.C = 1$ ):

- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the backward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{1,0,0\}$ .

For CPYMT, option A (encoded by  $PSTATE.C = 0$ ), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number.
- If the copy is in the forward direction ( $X_n$  is a negative number), then:
  - $X_n$  holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
  - $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
  - $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
  - At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- If the copy is in the backward direction ( $X_n$  is a positive number), then:
  - $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$  holds the highest address that the copy is copied from  $-X_n + 1$ .
  - $X_d$  holds the highest address that the copy is copied to  $-X_n + 1$ .
  - At the end of the instruction, the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMT, option B (encoded by  $PSTATE.C = 1$ ), the format of the arguments is:

- $X_n$  holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYET, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1\* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYET, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

## Integer

(FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sz		0		1		1		1		0		1		op1		0		Rs				0		0		1		1		0		1		Rn				Rd			
																op2																									

## Epilogue (op1 == 10)

```
CPYET [<Xd>]!, [<Xs>]!, <Xn>!
```

## Main (op1 == 01)

```
CPYMT [<Xd>]!, [<Xs>]!, <Xn>!
```

## Prologue (op1 == 00)

```
CPYPT [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFFFF<63:0>;

    boolean forward = IsMemCpyForward(toaddress, fromaddress, cpysize);

    if implements_option_a then
        nzcvc = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                nzcvc = '1010';
            else
                nzcvc = '0010';
        else
            CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            if stage != MOPSSStage\_Prologue then
                X[n, 64] = cpysize;
        else
            while UInt(stagecpysize) > 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= UInt(stagecpysize);

```

```

if nzcv<3> == '0' then // PSTATE.N
    readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
    Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
else
    readdata<B*8-1:0> = Mem[fromaddress-B, B, raccdesc];
    Mem[toaddress-B, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress - B;
    toaddress = toaddress - B;

cpysize = cpysize - B;
stagecpysize = stagecpysize - B;

if stage != MOPSSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;

if stage == MOPSSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYPTN, CPYMTN, CPYETN

Memory Copy, reads and writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTN, then CPYMTN, and then CPYETN.

CPYPTN performs some preconditioning of the arguments suitable for using the CPYMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYETN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPTN, the following saturation logic is applied:

If  $X_n < 63:55 \neq 000000000$ , the copy size  $X_n$  is saturated to  $0x007FFFFFFFFFFFFFFF$ .

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If  $(X_s > X_d) \&\& (X_d + \text{saturated } X_n) > X_s$ , then direction = forward

Elsif  $(X_s < X_d) \&\& (X_s + \text{saturated } X_n) > X_d$ , then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPTN, option A (which results in encoding  $PSTATE.C = 0$ ):

- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n$ .
  - $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- If the copy is in the backward direction, then:
  - $X_s$  and  $X_d$  are unchanged.
  - $X_n$  holds the saturated value of  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .

After execution of CPYPTN, option B (which results in encoding  $PSTATE.C = 1$ ):

- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the backward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{1,0,0\}$ .

For CPYMTN, option A (encoded by  $PSTATE.C = 0$ ), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number.
- If the copy is in the forward direction ( $X_n$  is a negative number), then:
  - $X_n$  holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
  - $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
  - $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
  - At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- If the copy is in the backward direction ( $X_n$  is a positive number), then:
  - $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$  holds the highest address that the copy is copied from  $-X_n+1$ .
  - $X_d$  holds the highest address that the copy is copied to  $-X_n+1$ .
  - At the end of the instruction, the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMTN, option B (encoded by  $PSTATE.C = 1$ ), the format of the arguments is:

- $X_n$  holds the number of bytes to be copied in the memory copy in total.



- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1\* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

## Integer

(FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																		
sz		0		1		1		1		0		1		op1		0		Rs				1				1				1				1				0		1		Rn				Rd			
																op2																																	

## Epilogue (op1 == 10)

```
CPYETN [<Xd>]!, [<Xs>]!, <Xn>!
```

## Main (op1 == 01)

```
CPYMTN [<Xd>]!, [<Xs>]!, <Xn>!
```

## Prologue (op1 == 00)

```
CPYPTN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFFFF<63:0>;

    boolean forward = IsMemCpyForward(toaddress, fromaddress, cpysize);

    if implements_option_a then
        nzcvc = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                nzcvc = '1010';
            else
                nzcvc = '0010';
    else
        CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            if stage != MOPSSStage\_Prologue then
                X[n, 64] = cpysize;
    else
        while UInt(stagecpysize) > 0 do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(toaddress, fromaddress, cpysize);
            assert B <= UInt(stagecpysize);

```

```

if nzcv<3> == '0' then // PSTATE.N
    readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
    Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
else
    readdata<B*8-1:0> = Mem[fromaddress-B, B, raccdesc];
    Mem[toaddress-B, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress - B;
    toaddress = toaddress - B;

cpysize = cpysize - B;
stagecpysize = stagecpysize - B;

if stage != MOPSSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;

if stage == MOPSSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYPTRN, CPYMTRN, CPYETRN

Memory Copy, reads and writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTRN, then CPYMTRN, and then CPYETRN.

CPYPTRN performs some preconditioning of the arguments suitable for using the CPYMTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYETRN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPTRN, the following saturation logic is applied:

If  $X_n < 63:55 \neq 000000000$ , the copy size  $X_n$  is saturated to  $0x007FFFFFFFFFFFFFFF$ .

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If  $(X_s > X_d) \&\& (X_d + \text{saturated } X_n) > X_s$ , then direction = forward

Elseif  $(X_s < X_d) \&\& (X_s + \text{saturated } X_n) > X_d$ , then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPTRN, option A (which results in encoding  $PSTATE.C = 0$ ):

- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n$ .
  - $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- If the copy is in the backward direction, then:
  - $X_s$  and  $X_d$  are unchanged.
  - $X_n$  holds the saturated value of  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .

After execution of CPYPTRN, option B (which results in encoding  $PSTATE.C = 1$ ):

- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the backward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{1,0,0\}$ .

For CPYMTRN, option A (encoded by  $PSTATE.C = 0$ ), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number.
- If the copy is in the forward direction ( $X_n$  is a negative number), then:
  - $X_n$  holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
  - $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
  - $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
  - At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- If the copy is in the backward direction ( $X_n$  is a positive number), then:
  - $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$  holds the highest address that the copy is copied from  $-X_n+1$ .
  - $X_d$  holds the highest address that the copy is copied to  $-X_n+1$ .
  - At the end of the instruction, the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMTRN, option B (encoded by  $PSTATE.C = 1$ ), the format of the arguments is:

- $X_n$  holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYETRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1\* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYETRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

## Integer

(FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sz		0		1		1		1		0		1		op1		0		Rs				1		0		1		1		0		1		Rn				Rd			
																op2																									

## Epilogue (op1 == 10)

```
CPYETRN [<Xd>]!, [<Xs>]!, <Xn>!
```

## Main (op1 == 01)

```
CPYMTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

## Prologue (op1 == 00)

```
CPYPTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.





```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFFFF<63:0>;

    boolean forward = IsMemCpyForward(toaddress, fromaddress, cpysize);

    if implements_option_a then
        nzcvc = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                nzcvc = '1010';
            else
                nzcvc = '0010';
    else
        CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            if stage != MOPSSStage\_Prologue then
                X[n, 64] = cpysize;
    else
        while UInt(stagecpysize) > 0 do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(toaddress, fromaddress, cpysize);
            assert B <= UInt(stagecpysize);

```

```

if nzcv<3> == '0' then // PSTATE.N
    readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
    Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
else
    readdata<B*8-1:0> = Mem[fromaddress-B, B, raccdesc];
    Mem[toaddress-B, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress - B;
    toaddress = toaddress - B;

cpysize = cpysize - B;
stagecpysize = stagecpysize - B;

if stage != MOPSSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;

if stage == MOPSSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYPTWN, CPYMTWN, CPYETWN

Memory Copy, reads and writes unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTWN, then CPYMTWN, and then CPYETWN.

CPYPTWN performs some preconditioning of the arguments suitable for using the CPYMTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYETWN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPTWN, the following saturation logic is applied:

If  $X_n < 63:55 \neq 00000000$ , the copy size  $X_n$  is saturated to  $0x007FFFFFFFFFFFFFFF$ .

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If  $(X_s > X_d) \ \&\& \ (X_d + \text{saturated } X_n) > X_s$ , then direction = forward

Elsif  $(X_s < X_d) \ \&\& \ (X_s + \text{saturated } X_n) > X_d$ , then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPTWN, option A (which results in encoding  $PSTATE.C = 0$ ):

- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n$ .
  - $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- If the copy is in the backward direction, then:
  - $X_s$  and  $X_d$  are unchanged.
  - $X_n$  holds the saturated value of  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .

After execution of CPYPTWN, option B (which results in encoding  $PSTATE.C = 1$ ):

- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the backward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{1,0,0\}$ .

For CPYMTWN, option A (encoded by  $PSTATE.C = 0$ ), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number.
- If the copy is in the forward direction ( $X_n$  is a negative number), then:
  - $X_n$  holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
  - $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
  - $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
  - At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- If the copy is in the backward direction ( $X_n$  is a positive number), then:
  - $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$  holds the highest address that the copy is copied from  $-X_n+1$ .
  - $X_d$  holds the highest address that the copy is copied to  $-X_n+1$ .
  - At the end of the instruction, the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMTWN, option B (encoded by  $PSTATE.C = 1$ ), the format of the arguments is:

- $X_n$  holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYETWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1\* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYETWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

## Integer

(FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sz		0		1		1		1		0		1		op1		0		Rs				0		1		1		1		0		1		Rn				Rd			
																op2																									

## Epilogue (op1 == 10)

```
CPYETWN [<Xd>]!, [<Xs>]!, <Xn>!
```

## Main (op1 == 01)

```
CPYMTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

## Prologue (op1 == 00)

```
CPYPTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFFFF<63:0>;

    boolean forward = IsMemCpyForward(toaddress, fromaddress, cpysize);

    if implements_option_a then
        nzcvc = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                nzcvc = '1010';
            else
                nzcvc = '0010';
        else
            CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            if stage != MOPSSStage\_Prologue then
                X[n, 64] = cpysize;
        else
            while UInt(stagecpysize) > 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= UInt(stagecpysize);

```



```

if nzcv<3> == '0' then // PSTATE.N
    readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
    Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
else
    readdata<B*8-1:0> = Mem[fromaddress-B, B, raccdesc];
    Mem[toaddress-B, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress - B;
    toaddress = toaddress - B;

cpysize = cpysize - B;
stagecpysize = stagecpysize - B;

if stage != MOPSSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;

if stage == MOPSSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYPWN, CPYMWN, CPYEWN

Memory Copy, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWN, then CPYMWN, and then CPYEWN.

CPYPWN performs some preconditioning of the arguments suitable for using the CPYMWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWN, the following saturation logic is applied:

If  $X_n < 63:55 > \neq 000000000$ , the copy size  $X_n$  is saturated to  $0x007FFFFFFFFFFFFFFF$ .

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If  $(X_s > X_d) \&\& (X_d + \text{saturated } X_n) > X_s$ , then direction = forward

Elsif  $(X_s < X_d) \&\& (X_s + \text{saturated } X_n) > X_d$ , then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWN, option A (which results in encoding  $PSTATE.C = 0$ ):

- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n$ .
  - $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- If the copy is in the backward direction, then:
  - $X_s$  and  $X_d$  are unchanged.
  - $X_n$  holds the saturated value of  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .

After execution of CPYPWN, option B (which results in encoding  $PSTATE.C = 1$ ):

- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the backward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{1,0,0\}$ .

For CPYMWN, option A (encoded by  $PSTATE.C = 0$ ), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number.
- If the copy is in the forward direction ( $X_n$  is a negative number), then:
  - $X_n$  holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
  - $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
  - $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
  - At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- If the copy is in the backward direction ( $X_n$  is a positive number), then:
  - $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$  holds the highest address that the copy is copied from  $-X_n+1$ .
  - $X_d$  holds the highest address that the copy is copied to  $-X_n+1$ .
  - At the end of the instruction, the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWN, option B (encoded by  $PSTATE.C = 1$ ), the format of the arguments is:

- $X_n$  holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1\* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYEWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

## Integer

(FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sz		0		1		1		1		0		1		op1		0		Rs				0		1		0		0		0		1		Rn				Rd			
																op2																									

## Epilogue (op1 == 10)

CPYEWN [<Xd>]!, [<Xs>]!, <Xn>!

## Main (op1 == 01)

CPYMWN [<Xd>]!, [<Xs>]!, <Xn>!

## Prologue (op1 == 00)

CPYPWN [<Xd>]!, [<Xs>]!, <Xn>!

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFFFF<63:0>;

    boolean forward = IsMemCpyForward(toaddress, fromaddress, cpysize);

    if implements_option_a then
        nzcvc = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                nzcvc = '1010';
            else
                nzcvc = '0010';
        else
            CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            if stage != MOPSSStage\_Prologue then
                X[n, 64] = cpysize;
        else
            while UInt(stagecpysize) > 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= UInt(stagecpysize);

```

```

if nzcv<3> == '0' then // PSTATE.N
    readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
    Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
else
    readdata<B*8-1:0> = Mem[fromaddress-B, B, raccdesc];
    Mem[toaddress-B, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress - B;
    toaddress = toaddress - B;

cpysize = cpysize - B;
stagecpysize = stagecpysize - B;

if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYPWT, CPYMWT, CPYEWT

Memory Copy, writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWT, then CPYMWT, and then CPYEWT.

CPYPWT performs some preconditioning of the arguments suitable for using the CPYMWT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWT performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWT, the following saturation logic is applied:

If  $X_n < 63:55 > \neq 000000000$ , the copy size  $X_n$  is saturated to  $0x007FFFFFFFFFFFFFFF$ .

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If  $(X_s > X_d) \&\& (X_d + \text{saturated } X_n) > X_s$ , then direction = forward

Elsif  $(X_s < X_d) \&\& (X_s + \text{saturated } X_n) > X_d$ , then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWT, option A (which results in encoding  $PSTATE.C = 0$ ):

- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n$ .
  - $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- If the copy is in the backward direction, then:
  - $X_s$  and  $X_d$  are unchanged.
  - $X_n$  holds the saturated value of  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .

After execution of CPYPWT, option B (which results in encoding  $PSTATE.C = 1$ ):

- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the backward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{1,0,0\}$ .

For CPYMWT, option A (encoded by  $PSTATE.C = 0$ ), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number.
- If the copy is in the forward direction ( $X_n$  is a negative number), then:
  - $X_n$  holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
  - $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
  - $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
  - At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- If the copy is in the backward direction ( $X_n$  is a positive number), then:
  - $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$  holds the highest address that the copy is copied from  $-X_n+1$ .
  - $X_d$  holds the highest address that the copy is copied to  $-X_n+1$ .
  - At the end of the instruction, the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWT, option B (encoded by  $PSTATE.C = 1$ ), the format of the arguments is:

- $X_n$  holds the number of bytes to be copied in the memory copy in total.



- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1\* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYEWT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

## Integer

(FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sz		0		1		1		1		0		1		op1		0		Rs				0		0		0		1		0		1		Rn				Rd			
op2																																									

## Epilogue (op1 == 10)

```
CPYEWTT [<Xd>]!, [<Xs>]!, <Xn>!
```

## Main (op1 == 01)

```
CPYMWT [<Xd>]!, [<Xs>]!, <Xn>!
```

## Prologue (op1 == 00)

```
CPYPWT [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd>** For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs>** For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn>** For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFFFF<63:0>;

    boolean forward = IsMemCpyForward(toaddress, fromaddress, cpysize);

    if implements_option_a then
        nzcvc = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                nzcvc = '1010';
            else
                nzcvc = '0010';
        else
            CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            if stage != MOPSSStage\_Prologue then
                X[n, 64] = cpysize;
        else
            while UInt(stagecpysize) > 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= UInt(stagecpysize);

```

```

if nzcv<3> == '0' then // PSTATE.N
    readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
    Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
else
    readdata<B*8-1:0> = Mem[fromaddress-B, B, raccdesc];
    Mem[toaddress-B, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress - B;
    toaddress = toaddress - B;

cpysize = cpysize - B;
stagecpysize = stagecpysize - B;

if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYPWTN, CPYMWTN, CPYEWTN

Memory Copy, writes unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTN, then CPYMWTN, and then CPYEWTN.

CPYPWTN performs some preconditioning of the arguments suitable for using the CPYMWTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWTN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWTN, the following saturation logic is applied:

If  $X_n < 63:55 > \neq 000000000$ , the copy size  $X_n$  is saturated to  $0x007FFFFFFFFFFFFFFF$ .

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If  $(X_s > X_d) \&\& (X_d + \text{saturated } X_n) > X_s$ , then direction = forward

Elsif  $(X_s < X_d) \&\& (X_s + \text{saturated } X_n) > X_d$ , then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWTN, option A (which results in encoding  $PSTATE.C = 0$ ):

- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n$ .
  - $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- If the copy is in the backward direction, then:
  - $X_s$  and  $X_d$  are unchanged.
  - $X_n$  holds the saturated value of  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .

After execution of CPYPWTN, option B (which results in encoding  $PSTATE.C = 1$ ):

- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the backward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{1,0,0\}$ .

For CPYMWTN, option A (encoded by  $PSTATE.C = 0$ ), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number.
- If the copy is in the forward direction ( $X_n$  is a negative number), then:
  - $X_n$  holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
  - $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
  - $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
  - At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- If the copy is in the backward direction ( $X_n$  is a positive number), then:
  - $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$  holds the highest address that the copy is copied from  $-X_n+1$ .
  - $X_d$  holds the highest address that the copy is copied to  $-X_n+1$ .
  - At the end of the instruction, the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWTN, option B (encoded by  $PSTATE.C = 1$ ), the format of the arguments is:

- $X_n$  holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1\* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYEWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

## Integer

(FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0														
sz		0		1		1		1		0		1		op1		0		Rs						1		1		0		1		0		1		Rn						Rd			
																op2																													

## Epilogue (op1 == 10)

```
CPYEWNTN [<Xd>]!, [<Xs>]!, <Xn>!
```

## Main (op1 == 01)

```
CPYMWTN [<Xd>]!, [<Xs>]!, <Xn>!
```

## Prologue (op1 == 00)

```
CPYPWTN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.





```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFFFF<63:0>;

    boolean forward = IsMemCpyForward(toaddress, fromaddress, cpysize);

    if implements_option_a then
        nzcvc = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                nzcvc = '1010';
            else
                nzcvc = '0010';
        else
            CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            if stage != MOPSSStage\_Prologue then
                X[n, 64] = cpysize;
        else
            while UInt(stagecpysize) > 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= UInt(stagecpysize);

```

```

if nzcv<3> == '0' then // PSTATE.N
    readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
    Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
else
    readdata<B*8-1:0> = Mem[fromaddress-B, B, raccdesc];
    Mem[toaddress-B, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress - B;
    toaddress = toaddress - B;

cpysize = cpysize - B;
stagecpysize = stagecpysize - B;

if stage != MOPSSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;

if stage == MOPSSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYPWTRN, CPYMWTRN, CPYEWTRN

Memory Copy, writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTRN, then CPYMWTRN, and then CPYEWTRN.

CPYPWTRN performs some preconditioning of the arguments suitable for using the CPYMWTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWTRN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWTRN, the following saturation logic is applied:

If  $X_n < 63:55 > \neq 000000000$ , the copy size  $X_n$  is saturated to  $0x007FFFFFFFFFFFFFFF$ .

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If  $(X_s > X_d) \&\& (X_d + \text{saturated } X_n) > X_s$ , then direction = forward

Elsif  $(X_s < X_d) \&\& (X_s + \text{saturated } X_n) > X_d$ , then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWTRN, option A (which results in encoding  $PSTATE.C = 0$ ):

- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n$ .
  - $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- If the copy is in the backward direction, then:
  - $X_s$  and  $X_d$  are unchanged.
  - $X_n$  holds the saturated value of  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .

After execution of CPYPWTRN, option B (which results in encoding  $PSTATE.C = 1$ ):

- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the backward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{1,0,0\}$ .

For CPYMWTRN, option A (encoded by  $PSTATE.C = 0$ ), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number.
- If the copy is in the forward direction ( $X_n$  is a negative number), then:
  - $X_n$  holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
  - $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
  - $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
  - At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- If the copy is in the backward direction ( $X_n$  is a positive number), then:
  - $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$  holds the highest address that the copy is copied from  $-X_n+1$ .
  - $X_d$  holds the highest address that the copy is copied to  $-X_n+1$ .
  - At the end of the instruction, the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWTRN, option B (encoded by  $PSTATE.C = 1$ ), the format of the arguments is:

- $X_n$  holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1\* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYEWTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

## Integer

(FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sz		0		1		1		1		0		1		op1		0		Rs				1		0		0		1		0		1		Rn				Rd			
																op2																									

## Epilogue (op1 == 10)

CPYEWTRN [<Xd>]!, [<Xs>]!, <Xn>!

## Main (op1 == 01)

CPYMWTRN [<Xd>]!, [<Xs>]!, <Xn>!

## Prologue (op1 == 00)

CPYPWTRN [<Xd>]!, [<Xs>]!, <Xn>!

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFFFF<63:0>;

    boolean forward = IsMemCpyForward(toaddress, fromaddress, cpysize);

    if implements_option_a then
        nzcvc = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                nzcvc = '1010';
            else
                nzcvc = '0010';
    else
        CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            if stage != MOPSSStage\_Prologue then
                X[n, 64] = cpysize;
    else
        while UInt(stagecpysize) > 0 do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(toaddress, fromaddress, cpysize);
            assert B <= UInt(stagecpysize);

```



```

if nzcv<3> == '0' then // PSTATE.N
    readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
    Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
else
    readdata<B*8-1:0> = Mem[fromaddress-B, B, raccdesc];
    Mem[toaddress-B, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress - B;
    toaddress = toaddress - B;

cpysize = cpysize - B;
stagecpysize = stagecpysize - B;

if stage != MOPSSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;

if stage == MOPSSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CPYPWTWN, CPYMWTWN, CPYEWTWN

Memory Copy, writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTWN, then CPYMWTWN, and then CPYEWTWN.

CPYPWTWN performs some preconditioning of the arguments suitable for using the CPYMWTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWTWN performs the last part of the memory copy.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWTWN, the following saturation logic is applied:

If  $X_n < 63:55 \neq 000000000$ , the copy size  $X_n$  is saturated to  $0x007FFFFFFFFFFFFFFF$ .

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If  $(X_s > X_d) \&\& (X_d + \text{saturated } X_n) > X_s$ , then direction = forward

Elsif  $(X_s < X_d) \&\& (X_s + \text{saturated } X_n) > X_d$ , then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWTWN, option A (which results in encoding  $PSTATE.C = 0$ ):

- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n$ .
  - $X_n$  holds  $-1 * \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
- If the copy is in the backward direction, then:
  - $X_s$  and  $X_d$  are unchanged.
  - $X_n$  holds the saturated value of  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .

After execution of CPYPWTWN, option B (which results in encoding  $PSTATE.C = 1$ ):

- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the backward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $X_n$  holds the saturated  $X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied}$ .
  - $PSTATE.\{N,Z,V\}$  are set to  $\{1,0,0\}$ .

For CPYMWTWN, option A (encoded by  $PSTATE.C = 0$ ), the format of the arguments is:

- $X_n$  is treated as a signed 64-bit number.
- If the copy is in the forward direction ( $X_n$  is a negative number), then:
  - $X_n$  holds  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
  - $X_s$  holds the lowest address that the copy is copied from  $-X_n$ .
  - $X_d$  holds the lowest address that the copy is made to  $-X_n$ .
  - At the end of the instruction, the value of  $X_n$  is written back with  $-1 * \text{the number of bytes remaining to be copied in the memory copy in total}$ .
- If the copy is in the backward direction ( $X_n$  is a positive number), then:
  - $X_n$  holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$  holds the highest address that the copy is copied from  $-X_n+1$ .
  - $X_d$  holds the highest address that the copy is copied to  $-X_n+1$ .
  - At the end of the instruction, the value of  $X_n$  is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWTWN, option B (encoded by  $PSTATE.C = 1$ ), the format of the arguments is:

- $X_n$  holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1\* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYEWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

## Integer

(FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
sz		0		1		1		1		0		1		op1		0		Rs				0		1		0		1		0		1		Rn				Rd			
																op2																									

## Epilogue (op1 == 10)

CPYEWTWN [<Xd>]!, [<Xs>]!, <Xn>!

## Main (op1 == 01)

CPYMWTWN [<Xd>]!, [<Xs>]!, <Xn>!

## Prologue (op1 == 00)

CPYPWTWN [<Xd>]!, [<Xs>]!, <Xn>!

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rntemporal = options<3> == '1';
boolean wntemporal = options<2> == '1';

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.



```

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;
integer B;

boolean implements_option_a = CPYOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged, wnontemporal);

if stage == MOPSSStage\_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFFFF<63:0>;

    boolean forward = IsMemCpyForward(toaddress, fromaddress, cpysize);

    if implements_option_a then
        nzcvc = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                nzcvc = '1010';
            else
                nzcvc = '0010';
        else
            CheckMemCpyParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, fromaddress, cpysize);

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress, cpysize);

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            if stage != MOPSSStage\_Prologue then
                X[n, 64] = cpysize;
        else
            while UInt(stagecpysize) > 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= UInt(stagecpysize);

```

```

if nzcv<3> == '0' then // PSTATE.N
    readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
    Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
else
    readdata<B*8-1:0> = Mem[fromaddress-B, B, raccdesc];
    Mem[toaddress-B, B, waccdesc] = readdata<B*8-1:0>;

    fromaddress = fromaddress - B;
    toaddress = toaddress - B;

cpysize = cpysize - B;
stagecpysize = stagecpysize - B;

if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# CRC32B, CRC32H, CRC32W, CRC32X

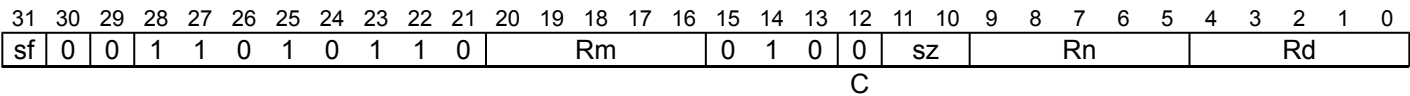
CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be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

In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

## Note

`ID_AA64ISAR0_EL1`.CRC32 indicates whether this instruction is supported.

## CRC (FEAT\_CRC32)



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

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

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

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

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

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

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

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

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

## Assembler Symbols

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



## Operation

```
bits(32) acc = X[n, 32];    // accumulator
bits(size) val = X[m, size]; // input value
bits(32) poly = 0x04C11DB7<31:0>;

bits(32+size) tempacc = BitReverse(acc):Zeros(size);
bits(size+32) tempval = BitReverse(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d, 32] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

## Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CRC32CB, CRC32CH, CRC32CW, CRC32CX

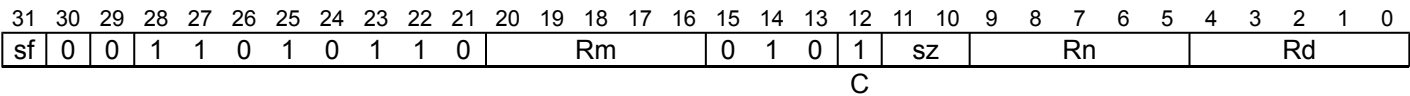
CRC32C checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x1EDC6F41 is used for the CRC calculation.

In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

Note

`ID_AA64ISAR0_EL1`.CRC32 indicates whether this instruction is supported.

CRC  
(FEAT\_CRC32)



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

CRC32CB <Wd>, <Wn>, <Wm>

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

CRC32CH <Wd>, <Wn>, <Wm>

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

CRC32CW <Wd>, <Wn>, <Wm>

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

CRC32CX <Wd>, <Wn>, <Xm>

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

Assembler Symbols

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

## Operation

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

bits(32+size) tempacc = BitReverse(acc):Zeros(size);
bits(size+32) tempval = BitReverse(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d, 32] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

## Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CSDB

Consumption of Speculative Data Barrier is a memory barrier that controls speculative execution and data value prediction.

No instruction other than branch instructions appearing in program order after the CSDB can be speculatively executed using the results of any:

- Data value predictions of any instructions.
- PSTATE.{N,Z,C,V} predictions of any instructions other than conditional branch instructions appearing in program order before the CSDB that have not been architecturally resolved.
- Predictions of SVE predication state for any SVE instructions.

Note

For purposes of the definition of CSDB, PSTATE.{N,Z,C,V} is not considered a data value. This definition permits:

- Control flow speculation before and after the CSDB.
- Speculative execution of conditional data processing instructions after the CSDB, unless they use the results of data value or PSTATE.{N,Z,C,V} predictions of instructions appearing in program order before the CSDB that have not been architecturally resolved.

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

CSDB

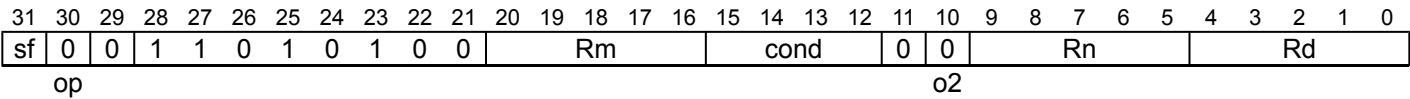
```
// Empty.
```

Operation

```
ConsumptionOfSpeculativeDataBarrier() ;
```

CSEL

If the condition is true, Conditional Select writes the value of the first source register to the destination register. If the condition is false, it writes the value of the second source register to the destination register.



32-bit (sf == 0)

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

64-bit (sf == 1)

```
CSEL <Xd>, <Xn>, <Xm>, <cond>
```

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

Assembler Symbols

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

Operation

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

Operational information

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

# CSET

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

This is an alias of [CSINC](#). This means:

- The encodings in this description are named to match the encodings of [CSINC](#).
- The description of [CSINC](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1	!= 111x			0	1	1	1	1	1	1	Rd					
op											Rm					cond			o2		Rn										

## 32-bit (sf == 0)

CSET <Wd>, <cond>

is equivalent to

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

and is always the preferred disassembly.

## 64-bit (sf == 1)

CSET <Xd>, <cond>

is equivalent to

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

and is always the preferred disassembly.

## Assembler Symbols

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

## Operation

The description of [CSINC](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CSETM

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

This is an alias of CSINV. This means:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		1	0	1	1	0	1	0	1	0	0	1	1	1	1	1	!= 111x				0	0	1	1	1	1	1	Rd			
op				Rm										cond				o2		Rn											

32-bit (sf == 0)

CSETM <Wd>, <cond>

is equivalent to

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

and is always the preferred disassembly.

64-bit (sf == 1)

CSETM <Xd>, <cond>

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

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

Operation

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

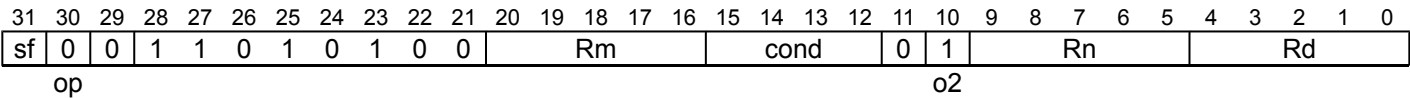
Operational information

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

CSINC

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

This instruction is used by the aliases [CINC](#), and [CSET](#).



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

Alias Conditions

Alias	Is preferred when
<a href="#">CINC</a>	Rm != '11111' && cond != '111x' && Rn != '11111' && Rn == Rm
<a href="#">CSET</a>	Rm == '11111' && cond != '111x' && Rn == '11111'

Operation

```
bits(datasize) result;
if ConditionHolds(cond) then
    result = X[n, datasize];
else
    result = X[m, datasize];
    result = result + 1;
X[d, datasize] = result;
```

Operational information

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



- The values of the NZCV flags.

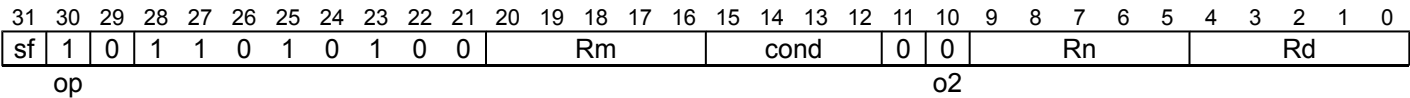
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CSINV

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

This instruction is used by the aliases [CINV](#), and [CSETM](#).



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

Alias Conditions

Alias	Is preferred when
<a href="#">CINV</a>	Rm != '11111' && cond != '111x' && Rn != '11111' && Rn == Rm
<a href="#">CSETM</a>	Rm == '11111' && cond != '111x' && Rn == '11111'

Operation

```
bits(datasize) result;
if ConditionHolds(cond) then
    result = X[n, datasize];
else
    result = X[m, datasize];
    result = NOT(result);
X[d, datasize] = result;
```

Operational information

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

- The values of the NZCV flags.

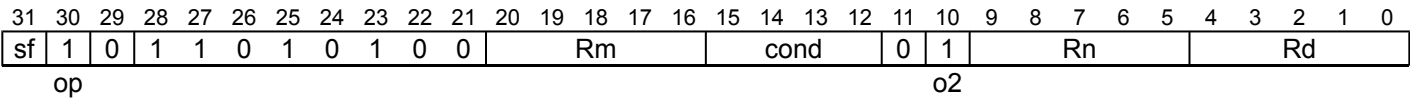
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CSNEG

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

This instruction is used by the alias [CNEG](#).



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

Alias Conditions

Alias	Is preferred when
<a href="#">CNEG</a>	cond != '111x' && Rn == Rm

Operation

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

Operational information

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

- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CTZ

Count Trailing Zeros counts the number of consecutive binary zero bits, starting from the least significant bit in the source register, and places the count in the destination register.

Integer  
(FEAT\_CSSC)

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

32-bit (sf == 0)

```
CTZ <Wd>, <Wn>
```

64-bit (sf == 1)

```
CTZ <Xd>, <Xn>
```

```
if !HaveCSSC() then UNDEFINED;
integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer d = UInt(Rd);
```

Assembler Symbols

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

Operation

```
bits(datasize) operand1 = X[n, datasize];
integer result = CountLeadingZeroBits(BitReverse(operand1));
X[d, datasize] = result<datasize-1:0>;
```

Operational information

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

DC

Data Cache operation. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions*.

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

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

DC [<dc\\_op>](#), [<Xt>](#)

is equivalent to

[SYS](#) [#<op1>](#), C7, [<Cm>](#), [#<op2>](#), [<Xt>](#)

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

Assembler Symbols

[<dc\\_op>](#) Is a DC instruction name, as listed for the DC system instruction group, encoded in “op1:CRm:op2”:

op1	CRm	op2	<dc_op>	Architectural Feature
000	0110	001	IVAC	-
000	0110	010	ISW	-
000	0110	011	IGVAC	FEAT_MTE2
000	0110	100	IGSW	FEAT_MTE2
000	0110	101	IGDVAC	FEAT_MTE2
000	0110	110	IGDSW	FEAT_MTE2
000	1010	010	CSW	-
000	1010	100	CGSW	FEAT_MTE2
000	1010	110	CGDSW	FEAT_MTE2
000	1110	010	CISW	-
000	1110	100	CIGSW	FEAT_MTE2
000	1110	110	CIGDSW	FEAT_MTE2
011	0100	001	ZVA	-
011	0100	011	GVA	FEAT_MTE
011	0100	100	GZVA	FEAT_MTE
011	1010	001	CVAC	-
011	1010	011	CGVAC	FEAT_MTE
011	1010	101	CGDVAC	FEAT_MTE
011	1011	001	CVAU	-
011	1100	001	CVAP	FEAT_DPB
011	1100	011	CGVAP	FEAT_MTE
011	1100	101	CGDVAP	FEAT_MTE
011	1101	001	CVADP	FEAT_DPB2
011	1101	011	CGVADP	FEAT_MTE
011	1101	101	CGDVADP	FEAT_MTE
011	1110	001	CIVAC	-
011	1110	011	CIGVAC	FEAT_MTE
011	1110	101	CIGDVAC	FEAT_MTE
100	1110	000	CIPAE	FEAT_MEC
100	1110	111	CIGDPAE	FEAT_MEC
110	1110	001	CIPAPA	FEAT_RME
110	1110	101	CIGDPAPA	FEAT_RME

[<op1>](#) Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op1” field.

[<Cm>](#) Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the “CRm” field.

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

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

## Operation

The description of [SYS](#) gives the operational pseudocode for this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



DCPS1

- Debug Change PE State to EL1, when executed in Debug state:
- If executed at EL0 changes the current Exception level and SP to EL1 using SP\_EL1.
  - Otherwise, if executed at ELx, selects SP\_ELx.

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

- When the target Exception level of a DCPS1 instruction is ELx, on executing this instruction:
- *ELR\_ELx* becomes UNKNOWN.
  - *SPSR\_ELx* becomes UNKNOWN.
  - *ESR\_ELx* becomes UNKNOWN.
  - *DLR\_EL0* and *DSPSR\_EL0* become UNKNOWN.
  - The endianness is set according to *SCTLR\_ELx*.EE.

This instruction is UNDEFINED at EL0 in Non-secure state if EL2 is implemented and *HCR\_EL2*.TGE == 1.  
This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPS<n> instructions, see *DCPS*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	1	0	1	imm16																0	0	0	0	1
LL																															

```
DCPS1 {#<imm>}
```

```
if !Halted() then UNDEFINED;
```

Assembler Symbols

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

Operation

```
DCPSInstruction (LL) ;
```

DCPS2

- Debug Change PE State to EL2, when executed in Debug state:
- If executed at EL0 or EL1 changes the current Exception level and SP to EL2 using SP\_EL2.
  - Otherwise, if executed at ELx, selects SP\_ELx.

- The target exception level of a DCPS2 instruction is:
- EL2 if the instruction is executed at an exception level that is not EL3.
  - EL3 if the instruction is executed at EL3.

- When the target Exception level of a DCPS2 instruction is ELx, on executing this instruction:
- *ELR\_ELx* becomes UNKNOWN.
  - *SPSR\_ELx* becomes UNKNOWN.
  - *ESR\_ELx* becomes UNKNOWN.
  - *DLR\_EL0* and *DSPSR\_EL0* become UNKNOWN.
  - The endianness is set according to *SCTLR\_ELx*.EE.

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

This instruction is always UNDEFINED in Non-debug state.  
For more information on the operation of the DCPS<n> instructions, see *DCPS*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	1	0	1	imm16																0	0	0	1	0
LL																															

```
DCPS2 {#<imm>}
```

```
if !Halted() then UNDEFINED;
```

Assembler Symbols

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

Operation

```
DCPSInstruction (LL) ;
```

DCPS3

- Debug Change PE State to EL3, when executed in Debug state:
- If executed at EL3 selects SP\_EL3.
  - Otherwise, changes the current Exception level and SP to EL3 using SP\_EL3.

The target exception level of a DCPS3 instruction is EL3.

- On executing a DCPS3 instruction:
- *ELR\_EL3* becomes UNKNOWN.
  - *SPSR\_EL3* becomes UNKNOWN.
  - *ESR\_EL3* becomes UNKNOWN.
  - *DLR\_EL0* and *DSPSR\_EL0* become UNKNOWN.
  - The endianness is set according to *SCTLR\_EL3*.EE.

- This instruction is UNDEFINED at all exception levels if either:
- *EDSCR*.SDD == 1.
  - EL3 is not implemented.

This instruction is always UNDEFINED in Non-debug state.  
For more information on the operation of the DCPS<n> instructions, see *DCPS*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	1	0	1	imm16																0	0	0	1	1
LL																															

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

```
if !Halted() then UNDEFINED;
```

Assembler Symbols

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

Operation

```
DCPSInstruction (LL) ;
```

DGH

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

System  
(FEAT\_DGH)

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

DGH

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

Operation

```
Hint_DGH();
```

DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see [Data Memory Barrier](#).

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

DMB <option>|<imm>

```
MBReqDomain domain;
MBReqTypes types;
case CRm<3:2> of
  when '00' domain = MBReqDomain_OuterShareable;
  when '01' domain = MBReqDomain_Nonshareable;
  when '10' domain = MBReqDomain_InnerShareable;
  when '11' domain = MBReqDomain_FullSystem;
case CRm<1:0> of
  when '00' types = MBReqTypes_All; domain = MBReqDomain_FullSystem;
  when '01' types = MBReqTypes_Reads;
  when '10' types = MBReqTypes_Writes;
  when '11' types = MBReqTypes_All;
```

Assembler Symbols

- <option> Specifies the limitation on the barrier operation. Values are:
- SY

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

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

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

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

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

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

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

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

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

**OSH**

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

**OSHST**

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

**OSHL**

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

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

<imm>

Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

**Operation**

```
DataMemoryBarrier(domain, types);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# DRPS

Debug restore process state

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

DRPS

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

## Operation

```
DRPSInstruction();
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42  
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see [Data Synchronization Barrier](#).  
A DSB instruction with the nXS qualifier is complete when the subset of these memory accesses with the XS attribute set to 0 are complete. It does not require that memory accesses with the XS attribute set to 1 are complete.  
This instruction is used by the aliases [PSSBB](#), and [SSBB](#).  
It has encodings from 2 classes: [Memory barrier](#) and [Memory nXS barrier](#)

Memory barrier

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

DSB <option>|<imm>

```
boolean nXS = FALSE;

DSBAlias alias;
case CRm of
  when '0000' alias = DSBAlias SSBB;
  when '0100' alias = DSBAlias PSSBB;
  otherwise alias = DSBAlias DSB;

MBReqDomain domain;
case CRm<3:2> of
  when '00' domain = MBReqDomain OuterShareable;
  when '01' domain = MBReqDomain Nonshareable;
  when '10' domain = MBReqDomain InnerShareable;
  when '11' domain = MBReqDomain FullSystem;

MBReqTypes types;
case CRm<1:0> of
  when '00' types = MBReqTypes All; domain = MBReqDomain FullSystem;
  when '01' types = MBReqTypes Reads;
  when '10' types = MBReqTypes Writes;
  when '11' types = MBReqTypes All;
```

Memory nXS barrier  
(FEAT\_XS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	imm2	1	0	0	0	1	1	1	1	1	1	0

DSB <option>nXS

```
if !HaveFeatXS() then UNDEFINED;
MBReqTypes types = MBReqTypes All;
boolean nXS = TRUE;
DSBAlias alias = DSBAlias DSB;
MBReqDomain domain;

case imm2 of
  when '00' domain = MBReqDomain OuterShareable;
  when '01' domain = MBReqDomain Nonshareable;
  when '10' domain = MBReqDomain InnerShareable;
  when '11' domain = MBReqDomain FullSystem;
```

Assembler Symbols

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



**SY**

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

**ST**

Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

**LD**

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

**ISH**

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

**ISHST**

Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

**ISHLD**

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

**NSH**

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

**NSHST**

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

**NSHLD**

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

**OSH**

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

**OSHST**

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

**OSHLD**

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

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

**Note**

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

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

**SY**

Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as imm2 = 0b11.

### ISH

Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as imm2 = 0b10.

### NSH

Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as imm2 = 0b01.

### OSH

Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as imm2 = 0b00.

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

## Alias Conditions

Alias	Is preferred when
<a href="#">PSSBB</a>	CRm == '0100'
<a href="#">SSBB</a>	CRm == '0000'

## Operation

```
case alias of
  when DSBAlias\_SSBB
    SpeculativeStoreBypassBarrierToVA();
  when DSBAlias\_PSSBB
    SpeculativeStoreBypassBarrierToPA();
  when DSBAlias\_DSB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure\_ERR, FALSE);
    if !nXS && HaveFeatXS() then
      nXS = PSTATE.EL IN {EL0, EL1} && IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1';
      DataSynchronizationBarrier(domain, types, nXS);
    otherwise
      Unreachable();
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# DVP

Data Value Prediction Restriction by Context prevents data value predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. Data value predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot be used to exploitatively control speculative execution occurring after the instruction is complete and synchronized.

For more information, see *DVP RCTX, Data Value Prediction Restriction by Context*.

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

## System (FEAT\_SPECRES)

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

DVP RCTX, [<Xt>](#)

is equivalent to

[SYS](#) #3, C7, C3, #5, [<Xt>](#)

and is always the preferred disassembly.

## Assembler Symbols

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

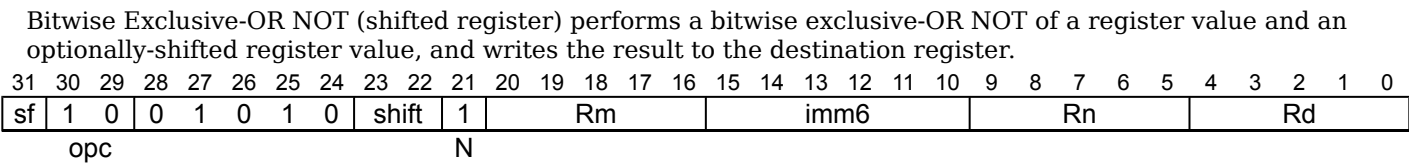
## Operation

The description of [SYS](#) gives the operational pseudocode for this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## EON (shifted register)



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

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

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

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

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

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

## Assembler Symbols

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

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR
- <amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.  
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

## Operation

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

operand2 = NOT(operand2);

result = operand1 EOR operand2;

X[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

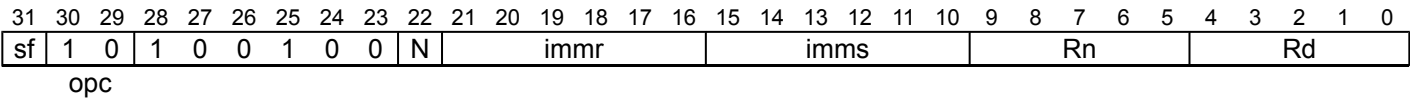
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EOR (immediate)

Bitwise Exclusive-OR (immediate) performs a bitwise exclusive-OR of a register value and an immediate value, and writes the result to the destination register.



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

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

Operation

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

result = operand1 EOR imm;

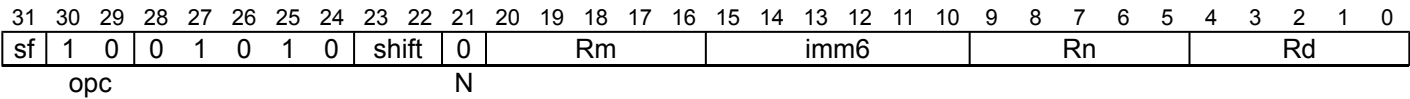
if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

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

EOR (shifted register)

Bitwise Exclusive-OR (shifted register) performs a bitwise exclusive-OR of a register value and an optionally-shifted register value, and writes the result to the destination register.



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

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

Assembler Symbols

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

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR
- <amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.  
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Operation

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

result = operand1 EOR operand2;

X[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



ERET

Exception Return using the ELR and SPSR for the current Exception level. When executed, the PE restores *PSTATE* from the SPSR, and branches to the address held in the ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state*.

ERET is UNDEFINED at EL0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
1	1	0	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0							
																					A		M	Rn								op4							

ERET

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

Operation

```
AArch64.CheckForERetTrap(FALSE, TRUE);
bits(64) target = ELR_ELx[];

AArch64.ExceptionReturn(target, SPSR_ELx[]);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ERETAA, ERETAB

Exception Return, with pointer authentication. This instruction authenticates the address in ELR, using SP as the modifier and the specified key, the PE restores *PSTATE* from the SPSR for the current Exception level, and branches to the authenticated address.

Key A is used for ERETAA. Key B is used for ERETAB.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see *Faulting on pointer authentication*.

The authenticated address is not written back to ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state*.

ERETAA and ERETAB are UNDEFINED at EL0.

Integer  
(FEAT\_PAuth)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	0	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	0	1	M	1	1	1	1	1	1	1	1	1	1				
																					A							Rn				op4			

ERETAA (M == 0)

ERETAA

ERETAB (M == 1)

ERETAB

```
if PSTATE.EL == EL0 then UNDEFINED;
boolean use_key_a = (M == '0');

if !HavePACExt() then
    UNDEFINED;
```

Operation

```
AArch64.CheckForERetTrap(TRUE, use_key_a);
bits(64) target = ELR_ELx[];
bits(64) modifier = SP[];

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

AArch64.ExceptionReturn(target, SPSR_ELx[]);
```

ESB

Error Synchronization Barrier is an error synchronization event that might also update DISR\_EL1 and VDISR\_EL2. This instruction can be used at all Exception levels and in Debug state. In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. See Error Synchronization Barrier in the Arm(R) Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for Armv8-A architecture profile. If the RAS Extension is not implemented, this instruction executes as a NOP.

System  
(FEAT\_RAS)

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

ESB

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

Operation

```
if HaveTME() && TSTATE.depth > 0 then
    FailTransaction(TMFailure_ERR, FALSE);
SynchronizeErrors();
AArch64.ESBOperation();
if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESBOperation();
TakeUnmaskedSErrorInterrupts();
```

EXTR

Extract register extracts a register from a pair of registers.  
This instruction is used by the alias [ROR \(immediate\)](#).

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

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

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

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

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

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

if N != sf then UNDEFINED;
if sf == '0' && imms<5> == '1' then UNDEFINED;
lsb = UInt(imms);
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <lsb> For the 32-bit variant: is the least significant bit position from which to extract, in the range 0 to 31, encoded in the "imms" field.  
For the 64-bit variant: is the least significant bit position from which to extract, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

Alias	Is preferred when
<a href="#">ROR (immediate)</a>	Rn == Rm

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
bits(2*datasize) concat = operand1:operand2;

result = concat<(lsb+datasize)-1:lsb>;

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# GCSB DSYNC

Guarded Control Stack Barrier. This instruction generates a Guarded control stack data synchronization event.  
If FEAT\_GCS is not implemented, this instruction executes as a NOP.

## System (FEAT\_GCS)

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

## GCSB DSYNC

```
if !IsFeatureImplemented(FEAT_GCS) then EndOfInstruction();
```

## Operation

```
GCSSynchronizationBarrier();
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42  
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

GCSPOPCX

Guarded Control Stack Pop and Compare exception return record loads an exception return record from the location indicated by the current Guarded control stack pointer register, compares the loaded values with the current ELR\_ELx, SPSR\_ELx, and LR, and increments the pointer by the size of a Guarded control stack exception return record.

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System  
(FEAT\_GCS)

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

GCSPOPCX {<Xt>}

is equivalent to

[SYS](#) #0, C7, C7, #5{, <Xt>}

and is always the preferred disassembly.

Assembler Symbols

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

The description of [SYS](#) gives the operational pseudocode for this instruction.

GCSPOPM

Guarded Control Stack Pop loads the 64-bit doubleword that is pointed to by the current Guarded control stack pointer, writes it to the destination register, and increments the current Guarded control stack pointer register by the size of a Guarded control stack procedure return record.

This is an alias of [SYSL](#). This means:

- The encodings in this description are named to match the encodings of [SYSL](#).
- The description of [SYSL](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System  
(FEAT\_GCS)

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

GCSPOPM <Xt>

is equivalent to

[SYSL](#) <Xt>, #3, C7, C7, #1

and is always the preferred disassembly.

Assembler Symbols

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

Operation

The description of [SYSL](#) gives the operational pseudocode for this instruction.



GCSPOPX

Guarded Control Stack Pop exception return record loads an exception return record from the location indicated by the current Guarded control stack pointer register, checks that the record is an exception return record, and increments the pointer by the size of a Guarded control stack exception return record.

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System  
(FEAT\_GCS)

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

GCSPOPX {<xt>}

is equivalent to

[SYS](#) #0, C7, C7, #6{, <xt>}

and is always the preferred disassembly.

Assembler Symbols

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

The description of [SYS](#) gives the operational pseudocode for this instruction.

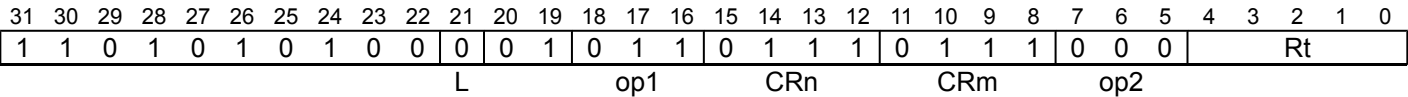
GCSPUSHM

Guarded Control Stack Push decrements the current Guarded control stack pointer register by the size of a Guarded control procedure return record and stores an entry to the Guarded control stack.

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System  
(FEAT\_GCS)



GCSPUSHM <Xt>

is equivalent to

[SYS](#) #3, C7, C7, #0, <Xt>

and is always the preferred disassembly.

Assembler Symbols

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

Operation

The description of [SYS](#) gives the operational pseudocode for this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# GCSPUSHX

Guarded Control Stack Push exception return record decrements the current Guarded control stack pointer register by the size of a Guarded control stack exception return record and stores an exception return record to the Guarded control stack.

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

## System (FEAT\_GCS)

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

GCSPUSHX {<Xt>}

is equivalent to

[SYS](#) #0, C7, C7, #4{, <Xt>}

and is always the preferred disassembly.

## Assembler Symbols

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

## Operation

The description of [SYS](#) gives the operational pseudocode for this instruction.

GCSSS1

Guarded Control Stack Switch Stack 1 validates that the stack being switched to contains a Valid cap entry, stores an In-progress cap entry to the stack that is being switched to, and sets the current Guarded control stack pointer to the stack that is being switched to.

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System  
(FEAT\_GCS)

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

GCSSS1 [<Xt>](#)

is equivalent to

[SYS](#) #3, C7, C7, #2, [<Xt>](#)

and is always the preferred disassembly.

Assembler Symbols

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

Operation

The description of [SYS](#) gives the operational pseudocode for this instruction.

GCSSS2

Guarded Control Stack Switch Stack 2 validates that the most recent entry of the Guarded control stack being switched to contains an In-progress cap entry, stores a Valid cap entry to the Guarded control stack that is being switched from, and sets Xt to the Guarded control stack pointer that is being switched from.

This is an alias of [SYSL](#). This means:

- The encodings in this description are named to match the encodings of [SYSL](#).
- The description of [SYSL](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System  
(FEAT\_GCS)

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

GCSSS2 <Xt>

is equivalent to

[SYSL](#) <Xt>, #3, C7, C7, #3

and is always the preferred disassembly.

Assembler Symbols

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

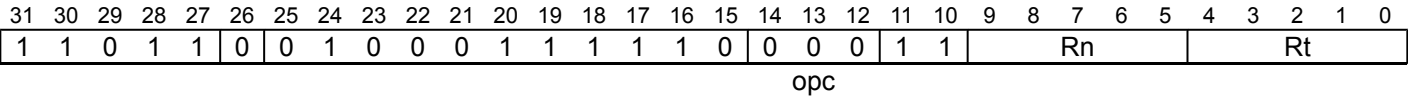
Operation

The description of [SYSL](#) gives the operational pseudocode for this instruction.

GCSSTR

Guarded Control Stack Store stores a doubleword from a register to memory. The address that is used for the store is calculated from a base register.

Integer  
(FEAT\_GCS)



```
GCSSTR <Xt>, [<Xn|SP>]
```

```
if !HaveGCS() then UNDEFINED;
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Assembler Symbols

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

Operation

```
bits(64) address;
bits(64) data;

bits(2) effective_el = PSTATE.EL;

if effective_el == PSTATE.EL then
    CheckGCSSTREnabled();

AccessDescriptor accdesc = CreateAccDescGCS(effective_el, MemOp_STORE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = X[t, 64];
Mem[address, 8, accdesc] = data;
```

GCSSTTR

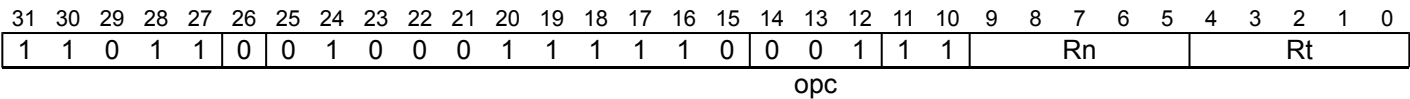
Guarded Control Stack unprivileged Store stores a doubleword from a register to memory. The address that is used for the store is calculated from a base register.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1 and *HCR\_EL2*.{NV, NV1} is not {1, 1}.
- The instruction is executed at EL2 when the *Effective value* of *HCR\_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed.

Integer  
(FEAT\_GCS)



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

```
if !HaveGCS() then UNDEFINED;
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Assembler Symbols

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

Operation

```
bits(64) address;
bits(64) data;

bits(2) effective_el = if AArch64.IsUnprivAccessPriv() then PSTATE.EL else EL0;

if effective_el == PSTATE.EL then
    CheckGCSSTREnabled();

AccessDescriptor accdesc = CreateAccDescGCS(effective_el, MemOp_STORE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = X[t, 64];
Mem[address, 8, accdesc] = data;
```

GMI

Tag Mask Insert inserts the tag in the first source register into the excluded set specified in the second source register, writing the new excluded set to the destination register.

Integer  
(FEAT\_MTE)

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

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

```
if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

Assembler Symbols

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

Operation

```
bits(64) address = if n == 31 then SP[] else X[n, 64];
bits(64) mask = X[m, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(address);

mask<UInt(tag)> = '1';
X[d, 64] = mask;
```



## HINT

Hint instruction is for the instruction set space that is reserved for architectural hint instructions.

Some encodings described here are not allocated in this revision of the architecture, and behave as NOPs. These encodings might be allocated to other hint functionality in future revisions of the architecture and therefore must not be used by software.

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

**HINT** #<imm>

```
SystemHintOp op;

case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0000 110'
    if !HaveDGHExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_DGH;
  when '0000 111' SEE "XPACLR1";
  when '0001 xxx'
    case op2 of
      when '000' SEE "PACIA1716";
      when '010' SEE "PACIB1716";
      when '100' SEE "AUTIA1716";
      when '110' SEE "AUTIB1716";
      otherwise EndOfInstruction();
  when '0010 000'
    if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_ESB;
  when '0010 001'
    if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_PSB;
  when '0010 010'
    if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_TSB;
  when '0010 011'
    if !IsFeatureImplemented(FEAT_GCS) then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_GCSB;
  when '0010 100'
    op = SystemHintOp_CSDB;
  when '0010 110'
    if !HaveFeatCLRBHB() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_CLRBHB;
  when '0011 xxx'
    case op2 of
      when '000' SEE "PACIAZ";
      when '001' SEE "PACIASP";
      when '010' SEE "PACIBZ";
      when '011' SEE "PACIBSP";
      when '100' SEE "AUTIAZ";
      when '101' SEE "AUTIASP";
      when '110' SEE "AUTIBZ";
      when '111' SEE "AUTIBSP";
  when '0100 xx0'
    op = SystemHintOp_BTI;
    // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
    SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
  when '0101 000'
    if !IsFeatureImplemented(FEAT_CHK) then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_CHKFEAT;
  otherwise EndOfInstruction();
```

## Assembler Symbols

<imm> Is a 7-bit unsigned immediate, in the range 0 to 127, encoded in the "CRm:op2" field. The encodings that are allocated to architectural hint functionality are described in the 'Hints' table in the 'Index by Encoding'.

### Note

For allocated encodings of "CRm:op2":

- A disassembler will disassemble the allocated instruction, rather than the `HINT` instruction.
- An assembler may support assembly of allocated encodings using `HINT` with the corresponding <imm> value, but it is not required to do so.

## Operation

```
case op of
  when SystemHintOp_YIELD
    Hint_Yield();

  when SystemHintOp_DGH
    Hint_DGH();

  when SystemHintOp_WFE
    integer localtimeout = 1 << 64;    // No local timeout event is generated
    Hint_WFE(localtimeout, WfxType_WFE);

  when SystemHintOp_WFI
    integer localtimeout = 1 << 64;    // No local timeout event is generated
    Hint_WFI(localtimeout, WfxType_WFI);

  when SystemHintOp_SEV
    SendEvent();

  when SystemHintOp_SEVL
    SendEventLocal();

  when SystemHintOp_ESB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure_ERR, FALSE);
      SynchronizeErrors();
      AArch64.ESBOperation();
      if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESBOperation();
      TakeUnmaskedSErrorInterrupts();

  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();

  when SystemHintOp_TSB
    TraceSynchronizationBarrier();

  when SystemHintOp_GCSB
    GCSSynchronizationBarrier();

  when SystemHintOp_CHKFEAT
    X[16, 64] = AArch64.ChkFeat(X[16, 64]);

  when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();

  when SystemHintOp_CLRBHB
    Hint_CLRBHB();

  when SystemHintOp_BTI
    SetBTypeNext('00');

  when SystemHintOp_NOP
    return;    // do nothing

  otherwise
    Unreachable();
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# HLT

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	0	1	0	imm16										0	0	0	0	0						

HLT #<imm>

```
if EDSCR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
if HaveBTIExt() then
    SetBTypeCompatible(TRUE);
```

## Assembler Symbols

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

## Operation

```
FaultRecord fault = NoFault();
Halt(DebugHalt_HaltInstruction, FALSE, fault);
```

HVC

Hypervisor Call causes an exception to EL2. Software executing at EL1 can use this instruction to call the hypervisor to request a service.

The HVC instruction is UNDEFINED:

- When EL3 is implemented and *SCR\_EL3.HCE* is set to 0.
- When EL3 is not implemented and *HCR\_EL2.HCD* is set to 1.
- When EL2 is not implemented.
- At EL1 if EL2 is not enabled in the current Security state.
- At EL0.

On executing an HVC instruction, the PE records the exception as a Hypervisor Call exception in *ESR\_ELx*, using the EC value 0x16, and the value of the immediate argument.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	0	0	0	imm16																0	0	0	1	0

HVC #<imm>

// Empty.

Assembler Symbols

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

Operation

```
if !HaveEL(EL2) || PSTATE.EL == EL0 || (PSTATE.EL == EL1 && !EL2Enabled()) then
    UNDEFINED;

bits(1) hvc_enable = if HaveEL(EL3) then SCR_EL3.HCE else NOT(HCR_EL2.HCD);

if hvc_enable == '0' then
    UNDEFINED;
else
    AArch64.CallHypervisor(imm16);
```

IC

Instruction Cache operation. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions*.

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

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

IC [<ic\\_op>](#){, [<Xt>](#)}

is equivalent to

[SYS](#) #[<op1>](#), C7, [<Cm>](#), #[<op2>](#){, [<Xt>](#)}

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

Assembler Symbols

[<ic\\_op>](#)

Is an IC instruction name, as listed for the IC system instruction pages, encoded in “op1:CRm:op2”:

op1	CRm	op2	<ic_op>
000	0001	000	IALLUIS
000	0101	000	IALLU
011	0101	001	IVAU

[<op1>](#)

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

[<Cm>](#)

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

[<op2>](#)

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

[<Xt>](#)

Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

The description of [SYS](#) gives the operational pseudocode for this instruction.

IRG

Insert Random Tag inserts a random Logical Address Tag into the address in the first source register, and writes the result to the destination register. Any tags specified in the optional second source register or in GCR\_EL1.Exclude are excluded from the selection of the random Logical Address Tag.

Integer  
(FEAT\_MTE)

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

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

```
if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

Assembler Symbols

- <Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.
- <Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Xm" field. Defaults to XZR if absent.

Operation

```
bits(64) operand = if n == 31 then SP[] else X[n, 64];
bits(64) exclude_reg = X[m, 64];
bits(16) exclude = exclude_reg<15:0> OR GCR_EL1.Exclude;
bits(4) rtag;

if AArch64.AllocationTagAccessIsEnabled(PSTATE.EL) then
    if GCR_EL1.RRND == '1' then
        if IsOnes(exclude) then
            rtag = '0000';
        else
            rtag = ChooseRandomNonExcludedTag(exclude);
    else
        bits(4) start_tag = RGSr_EL1.TAG;
        bits(4) offset = AArch64.RandomTag();

        rtag = AArch64.ChooseNonExcludedTag(start_tag, offset, exclude);

        RGSr_EL1.TAG = rtag;
else
    rtag = '0000';

bits(64) result = AArch64.AddressWithAllocationTag(operand, rtag);

if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;
```

ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see [Instruction Synchronization Barrier \(ISB\)](#).

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

ISB {<option>|<imm>}

// No additional decoding required

Assembler Symbols

- <option>

Specifies an optional limitation on the barrier operation. Values are:  
**SY**  
Full system barrier operation, encoded as CRm = 0b1111. Can be omitted.  
  
All other encodings of "CRm" are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.
- <imm>

Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

Operation

```
InstructionSynchronizationBarrier();  
if HaveBRBExt() && BRBEBranchOnISB() then  
    BRBEISB();
```



LD64B

Single-copy Atomic 64-byte Load derives an address from a base register value, loads eight 64-bit doublewords from a memory location, and writes them to consecutive registers, Xt to X(t+7). The data that is loaded is atomic and is required to be 64-byte aligned.

Integer  
(FEAT\_LS64)

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

LD64B <Xt>, [<Xn|SP> {, #0}]

```
if !HaveFeatLS64() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Xt> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckLDST64BEnabled();

bits(512) data;
bits(64) address;
bits(64) value;

AccessDescriptor accdesc = CreateAccDescLS64(memop, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemLoad64B(address, accdesc);

for i = 0 to 7
    value = data<63+64*i:64*i>;
    if BigEndian(accdesc.acctype) then value = BigEndianReverse(value);
    X[t+i, 64] = value;
```

LDADD, LDADDA, LDADDAL, LDADDL

Atomic add on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDADDA and LDADDAL load from memory with acquire semantics.
- LDADDL and LDADDAL store to memory with release semantics.
- LDADD has neither acquire nor release semantics.

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

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

This instruction is used by the alias [STADD, STADDL](#).

Integer  
(FEAT\_LSE)

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

32-bit LDADD (size == 10 && A == 0 && R == 0)

```
LDADD <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDADDA (size == 10 && A == 1 && R == 0)

```
LDADDA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDADDAL (size == 10 && A == 1 && R == 1)

```
LDADDAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDADDL (size == 10 && A == 0 && R == 1)

```
LDADDL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDADD (size == 11 && A == 0 && R == 0)

```
LDADD <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDADDA (size == 11 && A == 1 && R == 0)

```
LDADDA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDADDAL (size == 11 && A == 1 && R == 1)

```
LDADDAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDADDL (size == 11 && A == 0 && R == 1)

```
LDADDL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STADD, STADDL</a>	A == '0' && Rt == '11111'

## Operation

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

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ADD, acquire, release, tagchecked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(datasize) comparevalue = bits(datasize) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

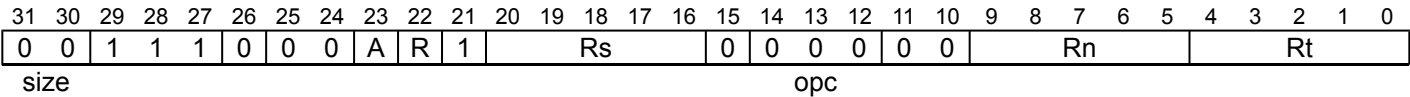
LDADDB, LDADDAB, LDADDALB, LDADDLB

Atomic add on byte in memory atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDADDAB and LDADDALB load from memory with acquire semantics.
- LDADDLB and LDADDALB store to memory with release semantics.
- LDADDB has neither acquire nor release semantics.

For more information about memory ordering semantics, see [Load-Acquire, Store-Release](#).  
For information about memory accesses, see [Load/Store addressing modes](#).  
This instruction is used by the alias [STADDB, STADDLB](#).

Integer  
(FEAT\_LSE)



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

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

LDADDALB (A == 1 && R == 1)

```
LDADDALB <Ws>, <Wt>, [<Xn|SP>]
```

LDADDB (A == 0 && R == 0)

```
LDADDB <Ws>, <Wt>, [<Xn|SP>]
```

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

```
LDADDLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

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

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STADDB, STADDLB</a>	A == '0' && Rt == '11111'

## Operation

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

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ADD, acquire, release, tagchecked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDADDH, LDADDAH, LDADDALH, LDADDLH

Atomic add on halfword in memory atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

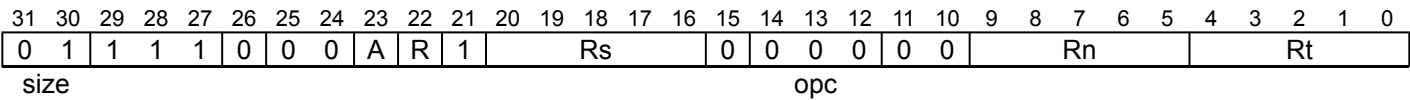
- If the destination register is not WZR, LDADDAH and LDADDALH load from memory with acquire semantics.
- LDADDLH and LDADDALH store to memory with release semantics.
- LDADDH has neither acquire nor release semantics.

For more information about memory ordering semantics, see [Load-Acquire, Store-Release](#).

For information about memory accesses, see [Load/Store addressing modes](#).

This instruction is used by the alias [STADDH, STADDLH](#).

## Integer (FEAT\_LSE)



### LDADDAH (A == 1 && R == 0)

```
LDADDAH <Ws>, <Wt>, [<Xn|SP>]
```

### LDADDALH (A == 1 && R == 1)

```
LDADDALH <Ws>, <Wt>, [<Xn|SP>]
```

### LDADDH (A == 0 && R == 0)

```
LDADDH <Ws>, <Wt>, [<Xn|SP>]
```

### LDADDLH (A == 0 && R == 1)

```
LDADDLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

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

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

## Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Alias Conditions

Alias	Is preferred when
<a href="#">STADDH, STADDLH</a>	A == '0' && Rt == '11111'

## Operation

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

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ADD, acquire, release, tagchecked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



LDAPR

Load-Acquire RCpc Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from the derived address in memory, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

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

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

No offset  
(FEAT\_LRCPC)

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

32-bit (size == 10)

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

64-bit (size == 11)

```
LDAPR <Xt>, [<Xn|SP> {, #0}]
```

```
boolean wback = FALSE;
integer offset = 0;
boolean wb_unknown = FALSE;

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

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
boolean tagchecked = n != 31;
```

Post-index  
(FEAT\_LRCPC3)

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

### 32-bit (size == 10)

```
LDAPR <Wt>, [<Xn|SP>], #4
```

### 64-bit (size == 11)

```
LDAPR <Xt>, [<Xn|SP>], #8
```

```
boolean wback = TRUE;

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

integer regsize = if size == '11' then 64 else 32;
integer datasize = 8 << UInt(size);
integer offset = 1 << UInt(size);

boolean tagchecked = TRUE;

boolean wb_unknown = FALSE;

if n == t && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint\_WBSUPPRESS, Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP};
    case c of
        when Constraint\_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint\_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint\_UNDEF      UNDEFINED;
        when Constraint\_NOP        EndOfInstruction();
```

## Assembler Symbols

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

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc = CreateAccDescLDAcqPC(tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    else
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDAPRB

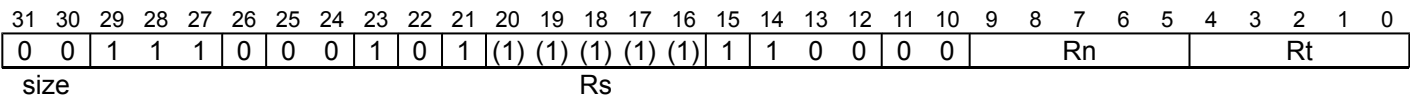
Load-Acquire RCpc Register Byte derives an address from a base register value, loads a byte from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.  
For information about memory accesses, see *Load/Store addressing modes*.

Integer  
(FEAT\_LRCPC)



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

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

boolean tagchecked = n != 31;
```

Assembler Symbols

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

Operation

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

AccessDescriptor accdesc = CreateAccDescLDAcqPC(tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

LDAPRH

Load-Acquire RCpc Register Halfword derives an address from a base register value, loads a halfword from the derived address in memory, zero-extends it and writes it to a register.

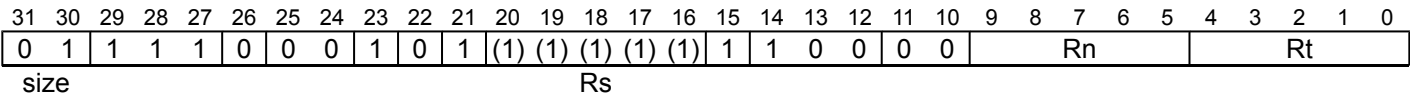
The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

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

Integer  
(FEAT\_LRCPC)



```
LDAPRH <Wt>, [<Xn|SP> {, #0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescLDAcqPC(tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDAPUR

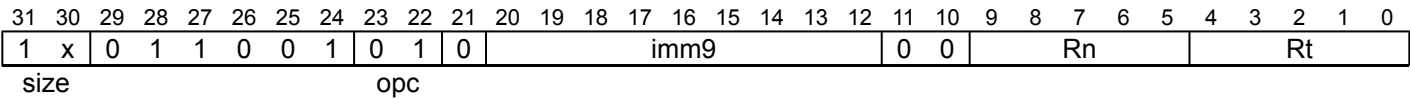
Load-Acquire RCpc Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.  
For information about memory accesses, see *Load/Store addressing modes*.

Unscaled offset  
(FEAT\_LRCPC2)



32-bit (size == 10)

```
LDAPUR <Wt>, [<Xn|SP>{, #<sim>}]
```

64-bit (size == 11)

```
LDAPUR <Xt>, [<Xn|SP>{, #<sim>}]
```

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tagchecked = n != 31;
```

## Operation

```
bits(64) address;
bits(datasize) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescLDAcqPC(tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDAPURB

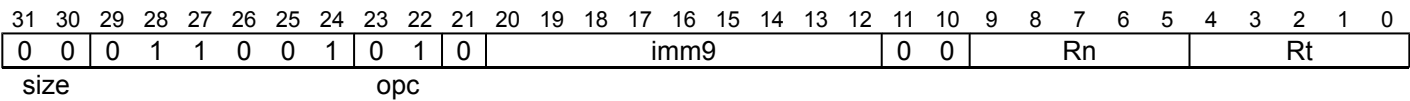
Load-Acquire RCpc Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.  
For information about memory accesses, see *Load/Store addressing modes*.

## Unscaled offset (FEAT\_LRCPC2)



```
LDAPURB <Wt>, [<Xn|SP>{, #<simm>}]

bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

## Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescLDAcqPC(tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.





# LDAPURH

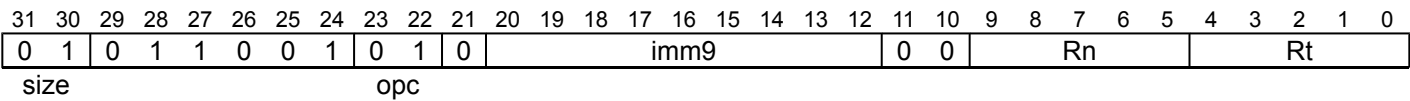
Load-Acquire RCpc Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.  
For information about memory accesses, see *Load/Store addressing modes*.

## Unscaled offset (FEAT\_LRCPC2)



```
LDAPURH <Wt>, [<Xn|SP>{, #<sim>}]

bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

## Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescLDAcqPC(tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



LDAPURSB

Load-Acquire RCpc Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register.

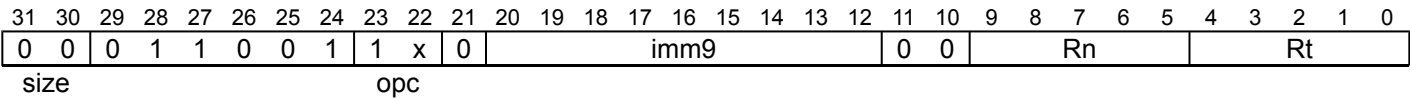
The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

Unscaled offset  
(FEAT\_LRCPC2)



32-bit (opc == 11)

```
LDAPURSB <Wt>, [<Xn|SP>{, #<sim>}]
```

64-bit (opc == 10)

```
LDAPURSB <Xt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp_PREFETCH && (n != 31);
```

## Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc;
if memop == MemOp_LOAD then
    accdesc = CreateAccDescLDacqPC(tagchecked);
elsif memop == MemOp_STORE then
    accdesc = CreateAccDescAcqRel(memop, tagchecked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t, 8];
        Mem[address, 1, accdesc] = data;

    when MemOp_LOAD
        data = Mem[address, 1, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDAPURSH

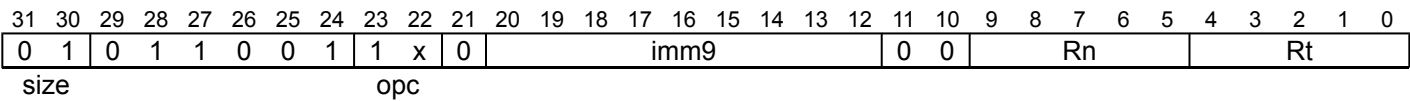
Load-Acquire RCpc Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.  
For information about memory accesses, see *Load/Store addressing modes*.

Unscaled offset  
(FEAT\_LRCPC2)



32-bit (opc == 11)

```
LDAPURSH <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (opc == 10)

```
LDAPURSH <Xt>, [<Xn|SP>{, #<simm>}]

bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp_PREFETCH && (n != 31);
```

## Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc;
if memop == MemOp_LOAD then
    accdesc = CreateAccDescLDacqPC(tagchecked);
elsif memop == MemOp_STORE then
    accdesc = CreateAccDescAcqRel(memop, tagchecked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t, 16];
        Mem[address, 2, accdesc] = data;

    when MemOp_LOAD
        data = Mem[address, 2, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDAPURSW

Load-Acquire RCpc Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register.

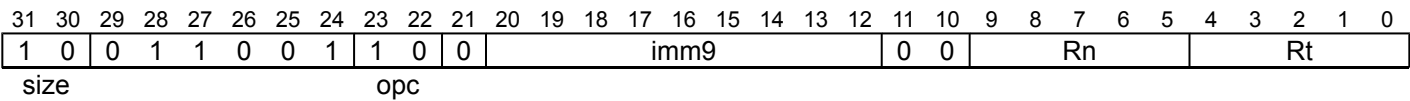
The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

## Unscaled offset (FEAT\_LRCPC2)



```
LDAPURSW <Xt>, [<Xn|SP>{, #<simm>}]

bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

## Operation

```
bits(64) address;
bits(32) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescLDAcqPC(tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 4, accdesc];
X[t, 64] = SignExtend(data, 64);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



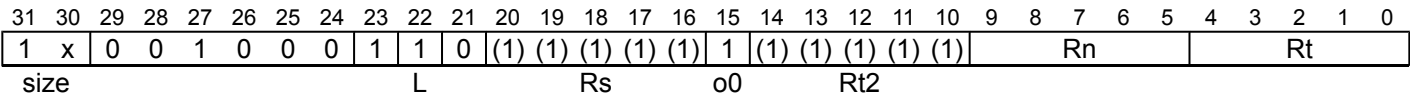


LDAR

Load-Acquire Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.



32-bit (size == 10)

```
LDAR <Wt>, [<Xn|SP>{, #0}]
```

64-bit (size == 11)

```
LDAR <Xt>, [<Xn|SP>{, #0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

AccessDescriptor accdesc;
accdesc = CreateAccDescAcqRel(MemOp_LOAD, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

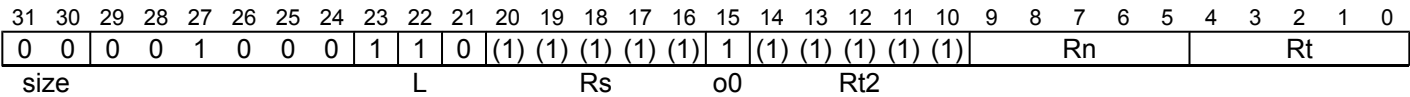
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

LDARB

Load-Acquire Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.



```
LDARB <Wt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescAcqRel(MemOp_LOAD, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

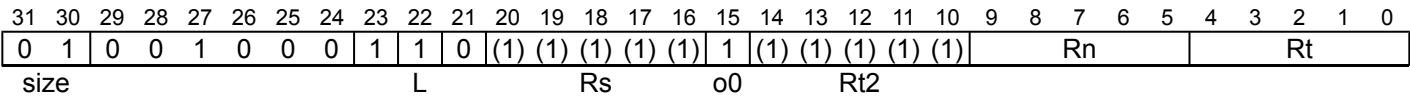
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

LDARH

Load-Acquire Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses, see [Load/Store addressing modes](#).

**Note**

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.



LDARH <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescAcqRel(MemOp_LOAD, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

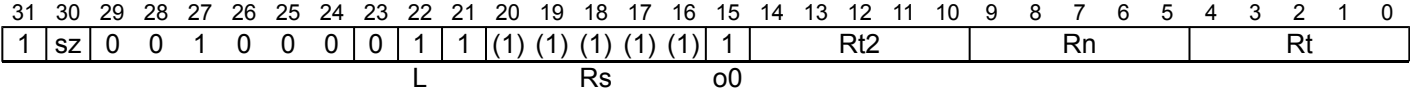
data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

# LDAXP

Load-Acquire Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information on single-copy atomicity and alignment requirements, see [Requirements for single-copy atomicity](#) and [Alignment of data accesses](#). The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). The instruction also has memory ordering semantics, as described in [Load-Acquire, Store-Release](#). For information about memory accesses, see [Load/Store addressing modes](#).



## 32-bit (sz == 0)

```
LDAXP <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

## 64-bit (sz == 1)

```
LDAXP <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDAXP](#).

## Assembler Symbols

- <Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, TRUE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t, datasize] = bits(datasize) UNKNOWN;    // In this case t = t2
elseif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, accdesc];
    if BigEndian(accdesc.acctype) then
        X[t, datasize-elsize] = data<datasize-1:elsize>;
        X[t2, elsize] = data<elsize-1:0>;
    else
        X[t, elsize] = data<elsize-1:0>;
        X[t2, datasize-elsize] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic), but must be 128-bit aligned
    if !IsAligned(address, dbytes) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    X[t, 64] = Mem[address, 8, accdesc];
    X[t2, 64] = Mem[address+8, 8, accdesc];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDAXR

Load-Acquire Exclusive Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs			o0			Rt2														

## 32-bit (size == 10)

```
LDAXR <Wt>, [<Xn|SP>{, #0}]
```

## 64-bit (size == 11)

```
LDAXR <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tagchecked = n != 31;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, TRUE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

data = Mem[address, dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.





LDAXRB

Load-Acquire Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn				Rt					
size								L		Rs				o0		Rt2															

```
LDAXRB <Wt>, [<Xn|SP>{, #0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, TRUE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

LDAXRH

Load-Acquire Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs				o0		Rt2														

```
LDAXRH <Wt>, [<Xn|SP>{, #0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, TRUE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

LDCLR, LDCLRA, LDCLRAL, LDCLRL

Atomic bit clear on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDCLRA and LDCLRAL load from memory with acquire semantics.
- LDCLRL and LDCLRAL store to memory with release semantics.
- LDCLR has neither acquire nor release semantics.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias [STCLR, STCLRL](#).

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	A	R	1	Rs				0	0	0	1	0	0	Rn				Rt						
size											opc																				

32-bit LDCLR (size == 10 && A == 0 && R == 0)

```
LDCLR <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDCLRA (size == 10 && A == 1 && R == 0)

```
LDCLRA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDCLRAL (size == 10 && A == 1 && R == 1)

```
LDCLRAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDCLRL (size == 10 && A == 0 && R == 1)

```
LDCLRL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDCLR (size == 11 && A == 0 && R == 0)

```
LDCLR <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDCLRA (size == 11 && A == 1 && R == 0)

```
LDCLRA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDCLRAL (size == 11 && A == 1 && R == 1)

```
LDCLRAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDCLRL (size == 11 && A == 0 && R == 1)

```
LDCLRL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STCLR, STCLRL</a>	A == '0' && Rt == '11111'

## Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire, release, tagchecked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(datasize) comparevalue = bits(datasize) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB

Atomic bit clear on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAB and LDCLRALB load from memory with acquire semantics.
- LDCLRLB and LDCLRALB store to memory with release semantics.
- LDCLRB has neither acquire nor release semantics.

For more information about memory ordering semantics, see [Load-Acquire, Store-Release](#).

For information about memory accesses, see [Load/Store addressing modes](#).

This instruction is used by the alias [STCLRB, STCLRLB](#).

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs				0	0	0	1	0	0	Rn				Rt						
size											opc																				

LDCLRAB (A == 1 && R == 0)

LDCLRAB <Ws>, <Wt>, [<Xn|SP>]

LDCLRALB (A == 1 && R == 1)

LDCLRALB <Ws>, <Wt>, [<Xn|SP>]

LDCLRB (A == 0 && R == 0)

LDCLRB <Ws>, <Wt>, [<Xn|SP>]

LDCLRLB (A == 0 && R == 1)

LDCLRLB <Ws>, <Wt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STCLRB, STCLRLB</a>	A == '0' && Rt == '11111'

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire, release, tagchecked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(8) comparevalue = bits(8) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH

Atomic bit clear on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAH and LDCLRALH load from memory with acquire semantics.
- LDCLRLH and LDCLRALH store to memory with release semantics.
- LDCLRH has neither acquire nor release semantics.

For more information about memory ordering semantics, see [Load-Acquire, Store-Release](#).

For information about memory accesses, see [Load/Store addressing modes](#).

This instruction is used by the alias [STCLRH, STCLRLH](#).

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs				0	0	0	1	0	0	Rn				Rt						
size											opc																				

LDCLRAH (A == 1 && R == 0)

LDCLRAH <Ws>, <Wt>, [<Xn|SP>]

LDCLRALH (A == 1 && R == 1)

LDCLRALH <Ws>, <Wt>, [<Xn|SP>]

LDCLRH (A == 0 && R == 0)

LDCLRH <Ws>, <Wt>, [<Xn|SP>]

LDCLRLH (A == 0 && R == 1)

LDCLRLH <Ws>, <Wt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STCLRH, STCLRLH</a>	A == '0' && Rt == '11111'



## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire, release, tagchecked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(16) comparevalue = bits(16) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

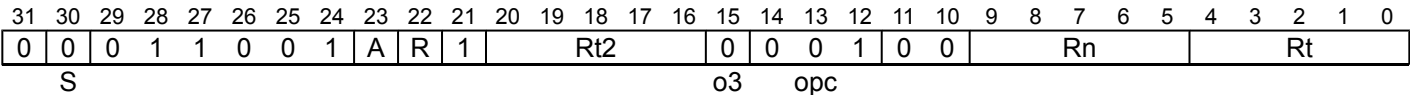
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL

Atomic bit clear on quadword in memory atomically loads a 128-bit quadword from memory, performs a bitwise AND with the complement of the value held in a pair of registers on it, and stores the result back to memory. The value initially loaded from memory is returned in the same pair of registers.

- LDCLRPA and LDCLRPAL load from memory with acquire semantics.
- LDCLRPL and LDCLRPAL store to memory with release semantics.
- LDCLRP has neither acquire nor release semantics.

## Integer (FEAT\_LSE128)



### LDCLRP (A == 0 && R == 0)

LDCLRP <Xt1>, <Xt2>, [<Xn|SP>]

### LDCLRPA (A == 1 && R == 0)

LDCLRPA <Xt1>, <Xt2>, [<Xn|SP>]

### LDCLRPAL (A == 1 && R == 1)

LDCLRPAL <Xt1>, <Xt2>, [<Xn|SP>]

### LDCLRPL (A == 0 && R == 1)

LDCLRPL <Xt1>, <Xt2>, [<Xn|SP>]

```
if !HaveLSE128() then UNDEFINED;
if Rt == '11111' then UNDEFINED;
if Rt2 == '11111' then UNDEFINED;
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer n = UInt(Rn);
boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable LSE128OVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint UNDEF    UNDEFINED;
        when Constraint NOP      EndOfInstruction();
```

## Assembler Symbols

- <Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(64) value1 = X[t, 64];
bits(64) value2 = X[t2, 64];
bits(128) data;
bits(128) store_value;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

store_value = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

bits(128) comparevalue = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

if rt_unknown then
    data = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = data<127:64>;
    X[t2, 64] = data<63:0>;
else
    X[t, 64] = data<63:0>;
    X[t2, 64] = data<127:64>;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDEOR, LDEORA, LDEORAL, LDEORL

Atomic Exclusive-OR on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDEORA and LDEORAL load from memory with acquire semantics.
- LDEORL and LDEORAL store to memory with release semantics.
- LDEOR has neither acquire nor release semantics.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias *STEOR, STEORL*.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	A	R	1	Rs				0	0	1	0	0	0	Rn				Rt						
size												opc																			

32-bit LDEOR (size == 10 && A == 0 && R == 0)

```
LDEOR <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDEORA (size == 10 && A == 1 && R == 0)

```
LDEORA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDEORAL (size == 10 && A == 1 && R == 1)

```
LDEORAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDEORL (size == 10 && A == 0 && R == 1)

```
LDEORL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDEOR (size == 11 && A == 0 && R == 0)

```
LDEOR <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDEORA (size == 11 && A == 1 && R == 0)

```
LDEORA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDEORAL (size == 11 && A == 1 && R == 1)

```
LDEORAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDEORL (size == 11 && A == 0 && R == 1)

```
LDEORL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STEOR</a> , <a href="#">STEORL</a>	A == '0' && Rt == '11111'

## Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_EOR, acquire, release, tagchecked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(datasize) comparevalue = bits(datasize) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDEORB, LDEORAB, LDEORALB, LDEORLB

Atomic Exclusive-OR on byte in memory atomically loads an 8-bit byte from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAB and LDEORALB load from memory with acquire semantics.
- LDEORLB and LDEORALB store to memory with release semantics.
- LDEORB has neither acquire nor release semantics.

For more information about memory ordering semantics, see [Load-Acquire, Store-Release](#).

For information about memory accesses, see [Load/Store addressing modes](#).

This instruction is used by the alias [STEORB, STEORLB](#).

## Integer (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs				0	0	1	0	0	0	Rn				Rt						
size											opc																				

### LDEORAB (A == 1 && R == 0)

LDEORAB <Ws>, <Wt>, [<Xn|SP>]

### LDEORALB (A == 1 && R == 1)

LDEORALB <Ws>, <Wt>, [<Xn|SP>]

### LDEORB (A == 0 && R == 0)

LDEORB <Ws>, <Wt>, [<Xn|SP>]

### LDEORLB (A == 0 && R == 1)

LDEORLB <Ws>, <Wt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Alias Conditions

Alias	Is preferred when
<a href="#">STEORB, STEORLB</a>	A == '0' && Rt == '11111'

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_EOR, acquire, release, tagchecked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(8) comparevalue = bits(8) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



LDEORH, LDEORAH, LDEORALH, LDEORLH

Atomic Exclusive-OR on halfword in memory atomically loads a 16-bit halfword from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAH and LDEORALH load from memory with acquire semantics.
- LDEORLH and LDEORALH store to memory with release semantics.
- LDEORH has neither acquire nor release semantics.

For more information about memory ordering semantics, see [Load-Acquire, Store-Release](#).

For information about memory accesses, see [Load/Store addressing modes](#).

This instruction is used by the alias [STEORH, STEORLH](#).

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs				0	0	1	0	0	0	Rn				Rt						
size											opc																				

LDEORAH (A == 1 && R == 0)

LDEORAH <Ws>, <Wt>, [<Xn|SP>]

LDEORALH (A == 1 && R == 1)

LDEORALH <Ws>, <Wt>, [<Xn|SP>]

LDEORH (A == 0 && R == 0)

LDEORH <Ws>, <Wt>, [<Xn|SP>]

LDEORLH (A == 0 && R == 1)

LDEORLH <Ws>, <Wt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STEORH, STEORLH</a>	A == '0' && Rt == '11111'

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_EOR, acquire, release, tagchecked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(16) comparevalue = bits(16) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

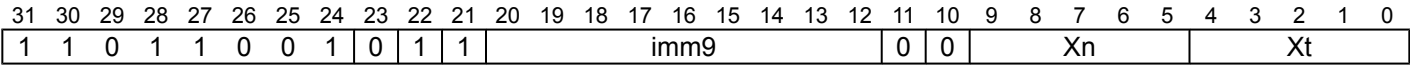
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDG

Load Allocation Tag loads an Allocation Tag from a memory address, generates a Logical Address Tag from the Allocation Tag and merges it into the destination register. The address used for the load is calculated from the base register and an immediate signed offset scaled by the Tag granule.

Integer  
(FEAT\_MTE)



LDG <Xt>, [<Xn|SP>{, #<sim>}]

```
if !HaveMTEExt() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
```

Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Xt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
- <sim> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

```
bits(64) address;
bits(4) tag;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;
address = Align(address, TAG_GRANULE);

AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_LOAD);
tag = AArch64.MemTag[address, accdesc];
X[t, 64] = AArch64.AddressWithAllocationTag(X[t, 64], tag);
```

LDGM

Load Tag Multiple reads a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID\_EL1.BS, and writes the Allocation Tag read from address A to the destination register at  $4 \times A \ll 7:4 + 3:4 \times A \ll 7:4$ . Bits of the destination register not written with an Allocation Tag are set to 0.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

Integer  
(FEAT\_MTE2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	Xn				Xt					

```
LDGM <Xt>, [<Xn|SP>]

if !HaveMTE2Ext() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);
```

Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Xt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

Operation

```
if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = Zeros(64);
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

integer size = 4 * (2 ^ (UInt(GMID_EL1.BS)));
address = Align(address, size);
integer count = size >> LOG2_TAG_GRANULE;
integer index = UInt(address < LOG2_TAG_GRANULE + 3: LOG2_TAG_GRANULE >);
AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_LOAD);

for i = 0 to count-1
    bits(4) tag = AArch64.MemTag[address, accdesc];
    data<(index*4)+3:index*4> = tag;
    address = address + TAG_GRANULE;
    index = index + 1;

X[t, 64] = data;
```

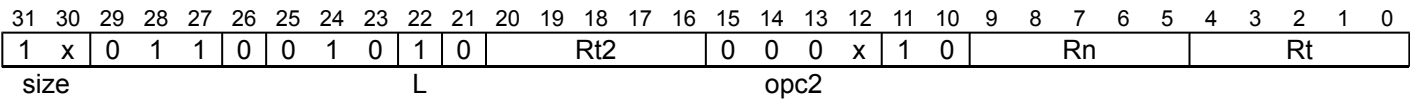
# LDIAPP

Load-Acquire RCpc ordered Pair of registers calculates an address from a base register value and an optional offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information on single-copy atomicity and alignment requirements, see *Requirements for single-copy atomicity* and *Alignment of data accesses*. The instruction also has memory ordering semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- The Memory effects associated with Xt1/Wt1 are Ordered-before the Memory effects associated with Xt2/Wt2.
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

For information about memory accesses, see *Load/Store addressing modes*.

## Integer (FEAT\_LRCPC3)



### 32-bit (size == 10 && opc2 == 0001)

```
LDIAPP <Wt1>, <Wt2>, [<Xn|SP>]
```

### 32-bit post-index (size == 10 && opc2 == 0000)

```
LDIAPP <Wt1>, <Wt2>, [<Xn|SP>], #8
```

### 64-bit (size == 11 && opc2 == 0001)

```
LDIAPP <Xt1>, <Xt2>, [<Xn|SP>]
```

### 64-bit post-index (size == 11 && opc2 == 0000)

```
LDIAPP <Xt1>, <Xt2>, [<Xn|SP>], #16
```

```
boolean postindex;
boolean wback;
postindex = opc2<0> == '0';
wback = opc2<0> == '0';
```

LDIAPP has the same CONSTRAINED UNPREDICTABLE behavior as LDP. For information about this CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDP*.

## Assembler Symbols

- <Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Shared Decode

```
integer offset;
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer scale = 2 + UInt(size<0>);
integer datasize = 8 << scale;
offset = if opc2<0> == '0' then (2 << scale) else 0;
boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
  assert c IN {Constraint WBSUPPRESS, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
  case c of
    when Constraint WBSUPPRESS wback = FALSE;      // writeback is suppressed
    when Constraint UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
    when Constraint UNDEF      UNDEFINED;
    when Constraint NOP        EndOfInstruction();

if t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable LDPOVERLAP);
  assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
  case c of
    when Constraint UNKNOWN rt_unknown = TRUE;      // result is UNKNOWN
    when Constraint UNDEF   UNDEFINED;
    when Constraint NOP     EndOfInstruction();
```

## Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_LOAD, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

if HaveLSE2Ext() then
    bits(2*datasize) full_data;
    accdesc.ispair = TRUE;
    full_data = Mem[address, 2*dbytes, accdesc];
    if BigEndian(accdesc.acctype) then
        data2 = full_data<(datasize-1):0>;
        data1 = full_data<(2*datasize-1):datasize>;
    else
        data1 = full_data<(datasize-1):0>;
        data2 = full_data<(2*datasize-1):datasize>;
else
    data1 = Mem[address, dbytes, accdesc];
    data2 = Mem[address+dbytes, dbytes, accdesc];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;

X[t, datasize] = data1;
X[t2, datasize] = data2;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDLAR

Load LOAcquire Register loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in [Load LOAcquire, Store LORelease](#). For information about memory accesses, see [Load/Store addressing modes](#).

## Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

## No offset (FEAT\_LOR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size				L					Rs					o0				Rt2													

## 32-bit (size == 10)

```
LDLAR <Wt>, [<Xn|SP>{, #0}]
```

## 64-bit (size == 11)

```
LDLAR <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tagchecked = n != 31;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

AccessDescriptor accdesc;
accdesc = CreateAccDescLOR(MemOp_LOAD, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.





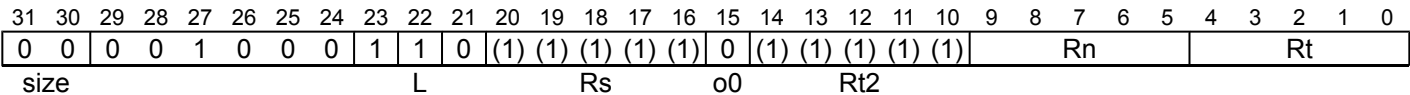
LDLARB

Load LOAcquire Register Byte loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

**Note**

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset  
(FEAT\_LOR)



```
LDLARB <Wt>, [<Xn|SP>{, #0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescLOR(MemOp_LOAD, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

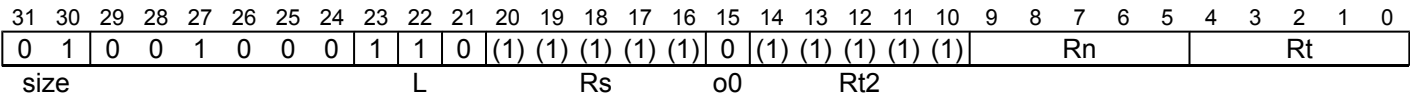
LDLARH

Load LOAcquire Register Halfword loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in [Load LOAcquire, Store LORelease](#). For information about memory accesses, see [Load/Store addressing modes](#).

**Note**

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset  
(FEAT\_LOR)



```
LDLARH <Wt>, [<Xn|SP>{, #0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescLOR(MemOp_LOAD, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

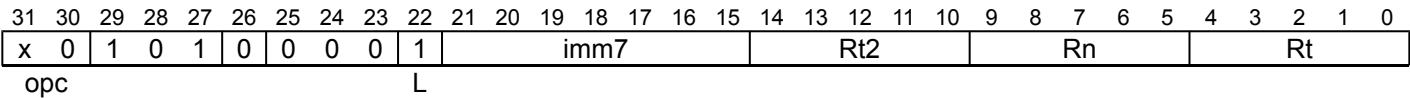
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDNP

Load Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about memory accesses, see [Load/Store addressing modes](#). For information about Non-temporal pair instructions, see [Load/Store Non-temporal pair](#).



32-bit (opc == 00)

```
LDNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 10)

```
LDNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
// Empty.
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDNP](#).

Assembler Symbols

- <Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.  
For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean privileged = PSTATE.EL != ELO;

AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_LOAD, TRUE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

if HaveLSE2Ext() then
    bits(2*datasize) full_data;
    accdesc.ispair = TRUE;
    full_data = Mem[address, 2*dbytes, accdesc];
    if BigEndian(accdesc.acctype) then
        data2 = full_data<(datasize-1):0>;
        data1 = full_data<(2*datasize-1):datasize>;
    else
        data1 = full_data<(datasize-1):0>;
        data2 = full_data<(2*datasize-1):datasize>;
else
    data1 = Mem[address, dbytes, accdesc];
    data2 = Mem[address+dbytes, dbytes, accdesc];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
X[t, datasize] = data1;
X[t2, datasize] = data2;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

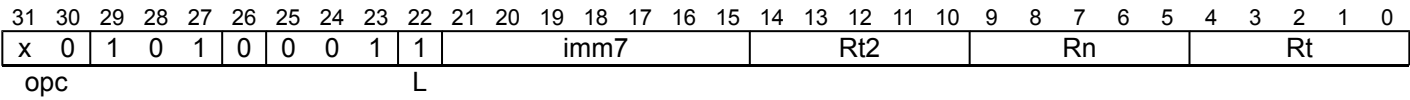
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDP

Load Pair of Registers calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Signed offset](#)

Post-index



32-bit (opc == 00)

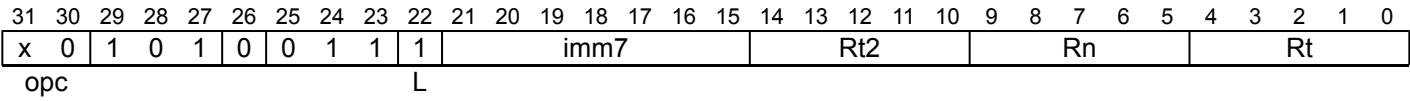
```
LDP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

64-bit (opc == 10)

```
LDP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index



32-bit (opc == 00)

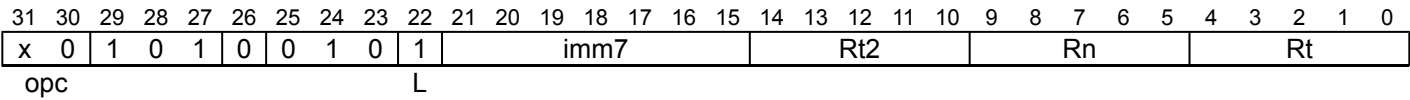
```
LDP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

64-bit (opc == 10)

```
LDP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset



32-bit (opc == 00)

```
LDP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 10)

```
LDP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDP](#).

## Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	<p>For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as &lt;imm&gt;/4.</p> <p>For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/4.</p> <p>For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as &lt;imm&gt;/8.</p> <p>For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as &lt;imm&gt;/8.</p>

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable WBOVERLAPLD);
    assert c IN {Constraint WBSUPPRESS, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint UNDEF UNDEFINED;
        when Constraint NOP EndOfInstruction();

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable LDPOVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint UNDEF UNDEFINED;
        when Constraint NOP EndOfInstruction();
```

## Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean privileged = PSTATE.EL != ELO;

AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

if HaveLSE2Ext() && !signed then
    bits(2*datasize) full_data;
    accdesc.ispair = TRUE;
    full_data = Mem[address, 2*dbytes, accdesc];
    if BigEndian(accdesc.acctype) then
        data2 = full_data<(datasize-1):0>;
        data1 = full_data<(2*datasize-1):datasize>;
    else
        data1 = full_data<(datasize-1):0>;
        data2 = full_data<(2*datasize-1):datasize>;
else
    data1 = Mem[address, dbytes, accdesc];
    data2 = Mem[address+dbytes, dbytes, accdesc];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
if signed then
    X[t, 64] = SignExtend(data1, 64);
    X[t2, 64] = SignExtend(data2, 64);
else
    X[t, datasize] = data1;
    X[t2, datasize] = data2;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

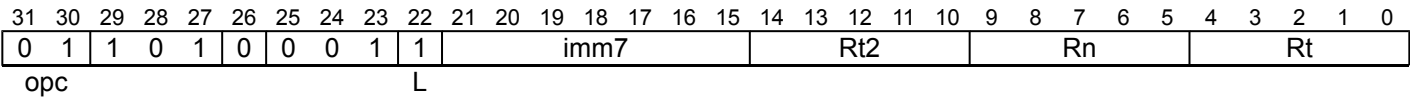


# LDPSW

Load Pair of Registers Signed Word calculates an address from a base register value and an immediate offset, loads two 32-bit words from memory, sign-extends them, and writes them to two registers. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Signed offset](#)

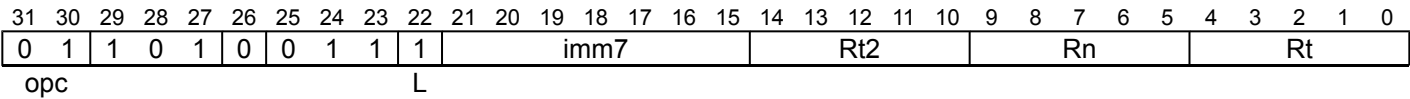
## Post-index



LDPSW <Xt1>, <Xt2>, [<Xn|SP>], #<imm>

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

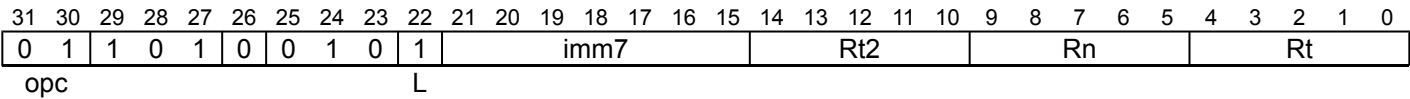
## Pre-index



LDPSW <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

## Signed offset



LDPSW <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDPSW](#).

## Assembler Symbols

- <Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the post-index and pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.  
For the signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
bits(64) offset = LSL(SignExtend(imm7, 64), 2);
boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable\_WBOVERLAPLD);
    assert c IN {Constraint\_WBSUPPRESS, Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP};
    case c of
        when Constraint\_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint\_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint\_UNDEF      UNDEFINED;
        when Constraint\_NOP        EndOfInstruction();

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable\_LDPOVERLAP);
    assert c IN {Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP};
    case c of
        when Constraint\_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint\_UNDEF    UNDEFINED;
        when Constraint\_NOP      EndOfInstruction();
```

## Operation

```
bits(64) address;
bits(32) data1;
bits(32) data2;
boolean privileged = PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

data1 = Mem[address, 4, accdesc];
data2 = Mem[address+4, 4, accdesc];
if rt_unknown then
    data1 = bits(32) UNKNOWN;
    data2 = bits(32) UNKNOWN;
X[t, 64] = SignExtend(data1, 64);
X[t2, 64] = SignExtend(data2, 64);
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



## LDR (immediate)

Load Register (immediate) loads a word or doubleword from memory and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#). The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value. It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	x	1	1	1	0	0	0	0	1	0	imm9									0	1	Rn				Rt									
size										opc																									

#### 32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>], #<sim>
```

#### 64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	x	1	1	1	0	0	0	0	1	0	imm9									1	1	Rn				Rt									
size										opc																									

#### 32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>, #<sim>]!
```

#### 64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	1	0	1	imm12												Rn				Rt					
size										opc																					

### 32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>{, #<pimm>}]
```

### 64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDR \(immediate\)](#).

### Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4. For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tagchecked = wback || n != 31;

boolean wb_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
case c of
  when Constraint_WBSUPPRESS wback = FALSE;      // writeback is suppressed
  when Constraint_UNKNOWN   wb_unknown = TRUE;    // writeback is UNKNOWN
  when Constraint_UNDEF     UNDEFINED;
  when Constraint_NOP       EndOfInstruction();
```

## Operation

```
bits(64) address;
bits(datasize) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

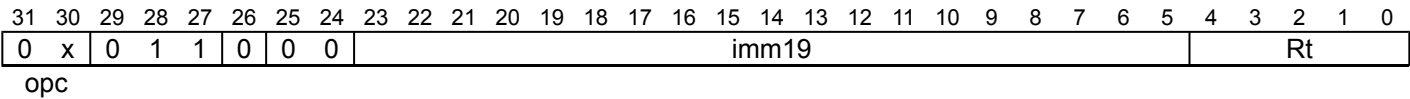
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



### 32-bit (opc == 00)

```
LDR <Wt>, <label>
```

### 64-bit (opc == 01)

```
LDR <Xt>, <label>
```

```
integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
  when '00'
    size = 4;
  when '01'
    size = 8;
  when '10'
    size = 4;
    signed = TRUE;
  when '11'
    memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

## Operation

```
bits(64) address = PC[] + offset;
bits(size*8) data;
boolean privileged = PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescGPR(memop, FALSE, privileged, FALSE);
case memop of
  when MemOp_LOAD
    data = Mem[address, size, accdesc];
    if signed then
      X[t, 64] = SignExtend(data, 64);
    else
      X[t, size*8] = data;

  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

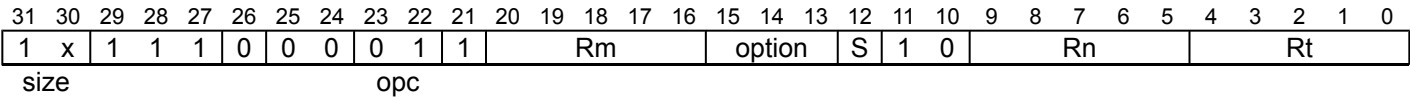
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about memory accesses, see [Load/Store addressing modes](#).



32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>, (<Wm|<Xm>){, <extend> {<amount>}}]
```

64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>, (<Wm|<Xm>){, <extend> {<amount>}}]
```

```
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- <extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SCTX

- <amount> For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#3

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(datasize) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDRAA, LDRAB

Load Register, with pointer authentication. This instruction authenticates an address from a base register using a modifier of zero and the specified key, adds an immediate offset to the authenticated address, and loads a 64-bit doubleword from memory at this resulting address into a register.

Key A is used for LDRAA. Key B is used for LDRAB.

If the authentication passes, the PE behaves the same as for an LDR instruction. For information on behavior if the authentication fails, see [Faulting on pointer authentication](#).

The authenticated address is not written back to the base register, unless the pre-indexed variant of the instruction is used. In this case, the address that is written back to the base register does not include the pointer authentication code.

For information about memory accesses, see [Load/Store addressing modes](#).

## Unscaled offset (FEAT\_PAuth)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	M	S	1	imm9									W	1	Rn				Rt					
size																															

### Key A, offset (M == 0 && W == 0)

```
LDRAA <Xt>, [<Xn|SP>{, #<simm>}]
```

### Key A, pre-indexed (M == 0 && W == 1)

```
LDRAA <Xt>, [<Xn|SP>{, #<simm>}]!
```

### Key B, offset (M == 1 && W == 0)

```
LDRAB <Xt>, [<Xn|SP>{, #<simm>}]
```

### Key B, pre-indexed (M == 1 && W == 1)

```
LDRAB <Xt>, [<Xn|SP>{, #<simm>}]!
```

```
if !HavePACExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
boolean wback = (W == '1');
boolean use_key_a = (M == '0');
bits(10) S10 = S:imm9;
bits(64) offset = LSL(SignExtend(S10, 64), 3);
boolean tagchecked = wback || n != 31;
```

## Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, a multiple of 8 in the range -4096 to 4088, defaulting to 0 and encoded in the "S:imm9" field as <simm>/8.

## Operation

```
bits(64) address;
bits(64) data;
boolean privileged = PSTATE.EL != EL0;
boolean wb_unknown = FALSE;

AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_LOAD, FALSE, privileged, tagchecked);
if wback && n == t && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable\_WBOVERLAPLD);
    assert c IN {Constraint\_WBSUPPRESS, Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP};
    case c of
        when Constraint\_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint\_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint\_UNDEF      UNDEFINED;
        when Constraint\_NOP        EndOfInstruction();

if n == 31 then
    address = SP[];
else
    address = X[n, 64];

if use_key_a then
    address = AuthDA(address, X[31, 64], TRUE);
else
    address = AuthDB(address, X[31, 64], TRUE);

if n == 31 then
    CheckSPAlignment();

address = address + offset;
data = Mem[address, 8, accdesc];
X[t, 64] = data;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

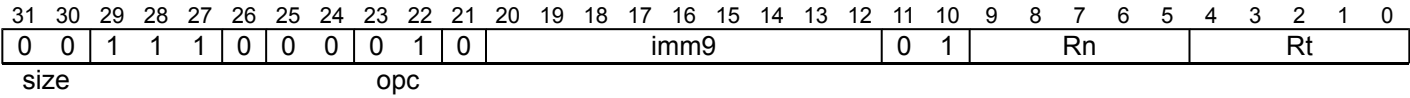
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDRB (immediate)

Load Register Byte (immediate) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Unsigned offset](#)

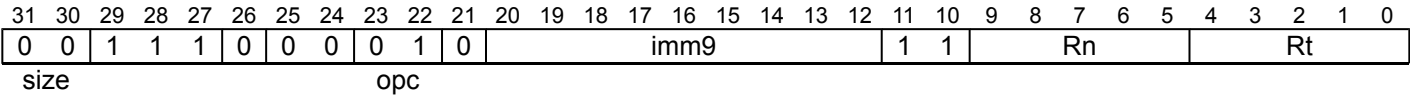
### Post-index



LDRB <Wt>, [<Xn|SP>], #<simm>

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

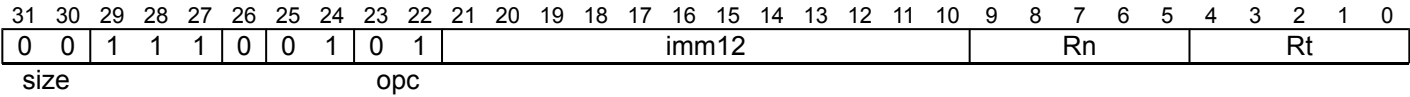
### Pre-index



LDRB <Wt>, [<Xn|SP>, #<simm>]!

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset



LDRB <Wt>, [<Xn|SP>{, #<pimm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDRH \(immediate\)](#).

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- <pimm> Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = wback || n != 31;

boolean wb_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstraintUnpredictable(Unpredictable\_WBOVERLAPLD);
    assert c IN {Constraint\_WBSUPPRESS, Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP};
    case c of
        when Constraint\_WBSUPPRESS wback = FALSE;      // writeback is suppressed
        when Constraint\_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint\_UNDEF      UNDEFINED;
        when Constraint\_NOP        EndOfInstruction();
```

## Operation

```
bits(64) address;
bits(8) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

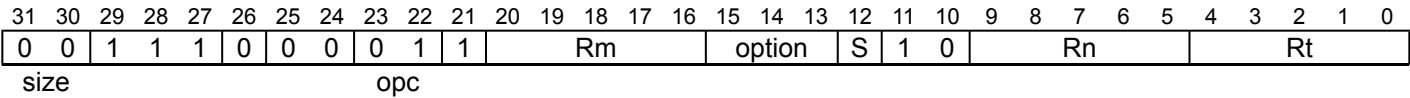
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



Extended register (option != 011)

```
LDRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Shifted register (option == 011)

```
LDRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend specifier, encoded in "option":

option	<extend>
010	UXTW
110	SXTW
111	SCTX

<amount> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0, 64);
bits(64) address;
bits(8) data;

boolean privileged = PSTATE.EL != ELO;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_LOAD, FALSE, privileged, TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

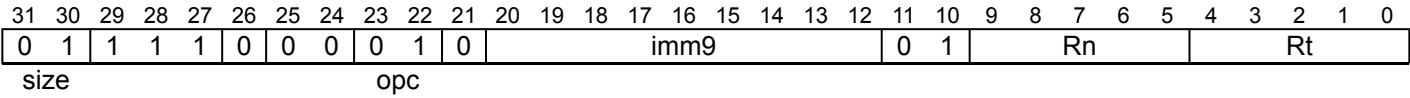


## LDRH (immediate)

Load Register Halfword (immediate) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Unsigned offset](#)

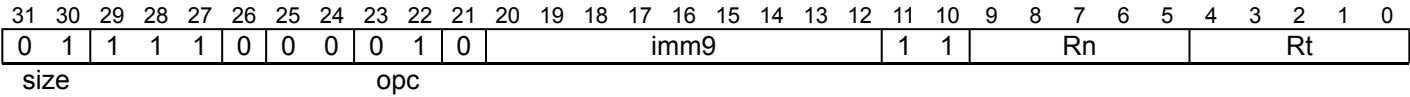
### Post-index



LDRH <Wt>, [<Xn|SP>], #<simm>

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

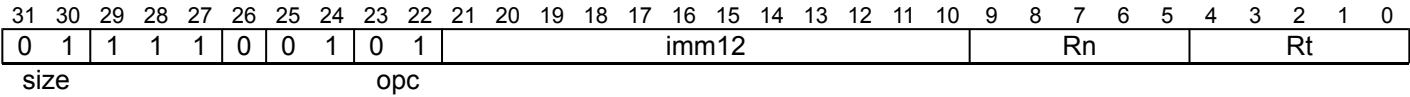
### Pre-index



LDRH <Wt>, [<Xn|SP>, #<simm>]!

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset



LDRH <Wt>, [<Xn|SP>{, #<pimm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDRH \(immediate\)](#).

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- <pimm> Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = wback || n != 31;

boolean wb_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstraintUnpredictable(Unpredictable\_WBOVERLAPLD);
    assert c IN {Constraint\_WBSUPPRESS, Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP};
    case c of
        when Constraint\_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint\_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint\_UNDEF      UNDEFINED;
        when Constraint\_NOP        EndOfInstruction();
```

## Operation

```
bits(64) address;
bits(16) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

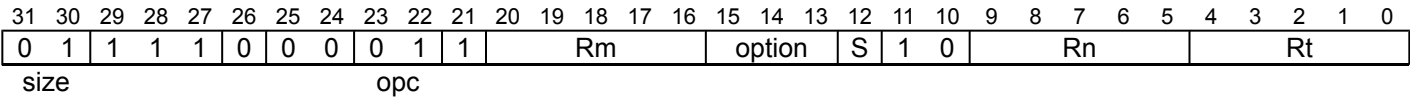
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



```
LDRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- <extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SCTX

- <amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#1

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(16) data;

boolean privileged = PSTATE.EL != ELO;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDRSB (immediate)

Load Register Signed Byte (immediate) loads a byte from memory, sign-extends it to either 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	0	1	1	1	0	0	0	1	x	0	imm9									0	1	Rn				Rt									
size										opc																									

#### 32-bit (opc == 11)

```
LDRSB <Wt>, [<Xn|SP>], #<sim>
```

#### 64-bit (opc == 10)

```
LDRSB <Xt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	0	1	1	1	0	0	0	1	x	0	imm9									1	1	Rn				Rt									
size										opc																									

#### 32-bit (opc == 11)

```
LDRSB <Wt>, [<Xn|SP>, #<sim>]!
```

#### 64-bit (opc == 10)

```
LDRSB <Xt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
0		0		1		1		1		0		0		1		1		x		imm12												Rn				Rt			
size										opc																													

32-bit (opc == 11)

```
LDRSB <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit (opc == 10)

```
LDRSB <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDRSB (immediate)*.

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- <pimm> Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp\_LOAD else MemOp\_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp\_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp\_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;
Constraint c;

if memop == MemOp\_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable\_WBOVERLAPLD);
    assert c IN {Constraint\_WBSUPPRESS, Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP};
    case c of
        when Constraint\_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint\_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint\_UNDEF      UNDEFINED;
        when Constraint\_NOP        EndOfInstruction();

if memop == MemOp\_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable\_WBOVERLAPST);
    assert c IN {Constraint\_NONE, Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP};
    case c of
        when Constraint\_NONE      rt_unknown = FALSE;    // value stored is original value
        when Constraint\_UNKNOWN    rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint\_UNDEF      UNDEFINED;
        when Constraint\_NOP        EndOfInstruction();
```

## Operation

```
bits(64) address;
bits(8) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(memop, FALSE, privileged, tagchecked);

if n == 31 then
    if memop != MemOp\_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

case memop of
    when MemOp\_STORE
        if rt_unknown then
            data = bits(8) UNKNOWN;
        else
            data = X[t, 8];
            Mem[address, 1, accdesc] = data;

    when MemOp\_LOAD
        data = Mem[address, 1, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp\_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

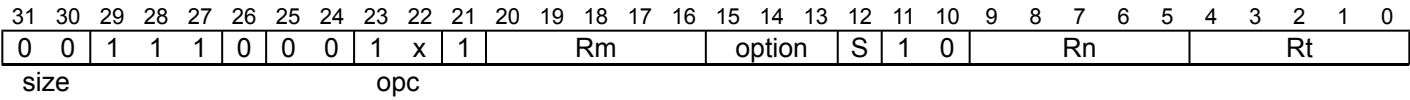
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



32-bit with extended register offset (opc == 11 && option != 011)

```
LDRSB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

32-bit with shifted register offset (opc == 11 && option == 011)

```
LDRSB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

64-bit with extended register offset (opc == 10 && option != 011)

```
LDRSB <Xt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

64-bit with shifted register offset (opc == 10 && option == 011)

```
LDRSB <Xt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend specifier, encoded in "option":

option	<extend>
010	UXTW
110	SXTW
111	SCTX

<amount> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp\_LOAD else MemOp\_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp\_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp\_PREFETCH;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0, 64);
bits(64) address;
bits(8) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(memop, FALSE, privileged, tagchecked);

if n == 31 then
    if memop != MemOp\_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp\_STORE
        data = X[t, 8];
        Mem[address, 1, accdesc] = data;

    when MemOp\_LOAD
        data = Mem[address, 1, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp\_PREFETCH
        Prefetch(address, t<4:0>);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDRSH (immediate)

Load Register Signed Halfword (immediate) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	1	1	1	1	0	0	0	1	x	0	imm9									0	1	Rn				Rt									
size										opc																									

#### 32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>], #<sim>
```

#### 64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	1	x	0	imm9									1	1	Rn				Rt					
size										opc																					

#### 32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>, #<sim>]!
```

#### 64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	1	1	x	imm12												Rn				Rt					
size										opc																					

**32-bit (opc == 11)**

```
LDRSH <Wt>, [<Xn|SP>{, #<pimm>}]
```

**64-bit (opc == 10)**

```
LDRSH <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDRSH (immediate)*.

**Assembler Symbols**

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- <pimm> Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp\_LOAD else MemOp\_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp\_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp\_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;
Constraint c;

if memop == MemOp\_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable\_WBOVERLAPLD);
    assert c IN {Constraint\_WBSUPPRESS, Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP};
    case c of
        when Constraint\_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint\_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint\_UNDEF      UNDEFINED;
        when Constraint\_NOP        EndOfInstruction();

if memop == MemOp\_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable\_WBOVERLAPST);
    assert c IN {Constraint\_NONE, Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP};
    case c of
        when Constraint\_NONE      rt_unknown = FALSE;    // value stored is original value
        when Constraint\_UNKNOWN    rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint\_UNDEF      UNDEFINED;
        when Constraint\_NOP        EndOfInstruction();
```

## Operation

```
bits(64) address;
bits(16) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(memop, FALSE, privileged, tagchecked);

if n == 31 then
    if memop != MemOp\_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

case memop of
    when MemOp\_STORE
        if rt_unknown then
            data = bits(16) UNKNOWN;
        else
            data = X[t, 16];
            Mem[address, 2, accdesc] = data;

    when MemOp\_LOAD
        data = Mem[address, 2, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp\_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

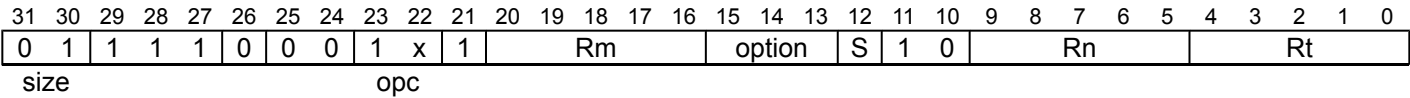
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- <extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SCTX

- <amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#1

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp\_LOAD else MemOp\_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp\_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp\_PREFETCH;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(16) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(memop, FALSE, privileged, tagchecked);

if n == 31 then
    if memop != MemOp\_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp\_STORE
        data = X[t, 16];
        Mem[address, 2, accdesc] = data;

    when MemOp\_LOAD
        data = Mem[address, 2, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp\_PREFETCH
        Prefetch(address, t<4:0>);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## LDRSW (immediate)

Load Register Signed Word (immediate) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	0	0	1	0	0	imm9									0	1	Rn				Rt					
size										opc																					

```
LDRSW <Xt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	0	0	1	0	0	imm9									1	1	Rn				Rt					
size										opc																					

```
LDRSW <Xt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	0	1	1	0	imm12												Rn				Rt					
size										opc																					

```
LDRSW <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 2);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDRSW \(immediate\)](#).

### Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- <pimm> Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = wback || n != 31;

boolean wb_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable\_WBOVERLAPLD);
    assert c IN {Constraint\_WBSUPPRESS, Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP};
    case c of
        when Constraint\_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint\_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint\_UNDEF      UNDEFINED;
        when Constraint\_NOP        EndOfInstruction();
```

## Operation

```
bits(64) address;
bits(32) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

data = Mem[address, 4, accdesc];
X[t, 64] = SignExtend(data, 64);
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDRSW (literal)

Load Register Signed Word (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	0	0	imm19												Rt											

opc

**LDRSW** <Xt>, <label>

```
integer t = UInt(Rt);
bits(64) offset;

offset = SignExtend(imm19:'00', 64);
```

### Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

### Operation

```
bits(64) address = PC[] + offset;
bits(32) data;
boolean privileged = PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, FALSE);
data = Mem[address, 4, accdesc];
X[t, 64] = SignExtend(data, 64);
```

### Operational information

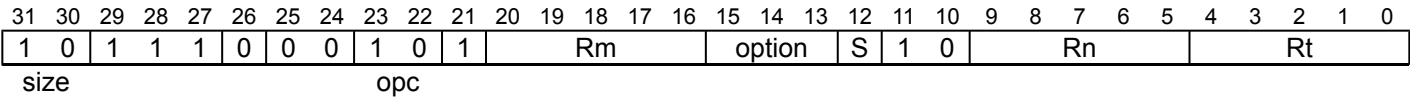
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSW (register)

Load Register Signed Word (register) calculates an address from a base register value and an offset register value, loads a word from memory, sign-extends it to form a 64-bit value, and writes it to a register. The offset register value can be shifted left by 0 or 2 bits. For information about memory accesses, see *Load/Store addressing modes*.



```
LDRSW <Xt>, [<Xn|SP>, (<Wm|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 2 else 0;
```

Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- <extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SCTX

- <amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#2

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(32) data;

boolean privileged = PSTATE.EL != ELO;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 4, accdesc];
X[t, 64] = SignExtend(data, 64);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDSET, LDSETA, LDSETAL, LDSETL

Atomic bit set on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSETA and LDSETAL load from memory with acquire semantics.
- LDSETL and LDSETAL store to memory with release semantics.
- LDSET has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias *STSET, STSETL*.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	A	R	1	Rs				0	0	1	1	0	0	Rn				Rt						
size												opc																			

32-bit LDSET (size == 10 && A == 0 && R == 0)

```
LDSET <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSETA (size == 10 && A == 1 && R == 0)

```
LDSETA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSETAL (size == 10 && A == 1 && R == 1)

```
LDSETAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSETL (size == 10 && A == 0 && R == 1)

```
LDSETL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDSET (size == 11 && A == 0 && R == 0)

```
LDSET <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSETA (size == 11 && A == 1 && R == 0)

```
LDSETA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSETAL (size == 11 && A == 1 && R == 1)

```
LDSETAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSETL (size == 11 && A == 0 && R == 1)

```
LDSETL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STSET, STSETL</a>	A == '0' && Rt == '11111'

## Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire, release, tagchecked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(datasize) comparevalue = bits(datasize) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



LDSETB, LDSETAB, LDSETALB, LDSETLB

Atomic bit set on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

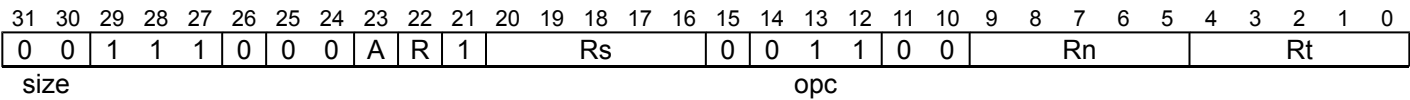
- If the destination register is not WZR, LDSETAB and LDSETALB load from memory with acquire semantics.
- LDSETLB and LDSETALB store to memory with release semantics.
- LDSETB has neither acquire nor release semantics.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias *STSETB, STSETLB*.

Integer  
(FEAT\_LSE)



LDSETAB (A == 1 && R == 0)

```
LDSETAB <Ws>, <Wt>, [<Xn|SP>]
```

LDSETALB (A == 1 && R == 1)

```
LDSETALB <Ws>, <Wt>, [<Xn|SP>]
```

LDSETB (A == 0 && R == 0)

```
LDSETB <Ws>, <Wt>, [<Xn|SP>]
```

LDSETLB (A == 0 && R == 1)

```
LDSETLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STSETB, STSETLB</a>	A == '0' && Rt == '11111'

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire, release, tagchecked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(8) comparevalue = bits(8) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDSETH, LDSETAH, LDSETALH, LDSETLH

Atomic bit set on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

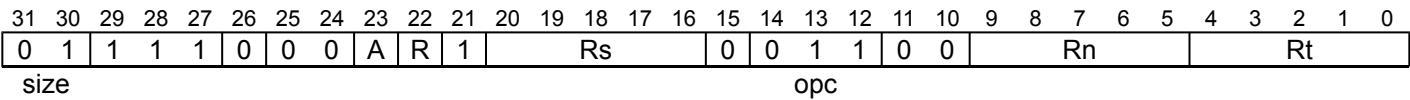
- If the destination register is not WZR, LDSETAH and LDSETALH load from memory with acquire semantics.
- LDSETLH and LDSETALH store to memory with release semantics.
- LDSETH has neither acquire nor release semantics.

For more information about memory ordering semantics, see [Load-Acquire, Store-Release](#).

For information about memory accesses, see [Load/Store addressing modes](#).

This instruction is used by the alias [STSETH, STSETLH](#).

Integer  
(FEAT\_LSE)



LDSETAH (A == 1 && R == 0)

```
LDSETAH <Ws>, <Wt>, [<Xn|SP>]
```

LDSETALH (A == 1 && R == 1)

```
LDSETALH <Ws>, <Wt>, [<Xn|SP>]
```

LDSETH (A == 0 && R == 0)

```
LDSETH <Ws>, <Wt>, [<Xn|SP>]
```

LDSETLH (A == 0 && R == 1)

```
LDSETLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STSETH, STSETLH</a>	A == '0' && Rt == '11111'

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire, release, tagchecked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

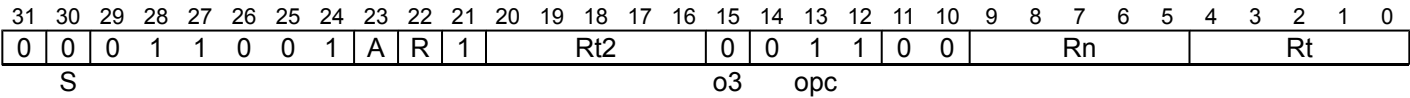
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDSETP, LDSETPA, LDSETPAL, LDSETPL

Atomic bit set on quadword in memory atomically loads a 128-bit quadword from memory, performs a bitwise OR with the value held in a pair of registers on it, and stores the result back to memory. The value initially loaded from memory is returned in the same pair of registers.

- LDSETPA and LDSETPAL load from memory with acquire semantics.
- LDSETPL and LDSETPAL store to memory with release semantics.
- LDSETP has neither acquire nor release semantics.

## Integer (FEAT\_LSE128)



### LDSETP (A == 0 && R == 0)

LDSETP <Xt1>, <Xt2>, [<Xn|SP>]

### LDSETPA (A == 1 && R == 0)

LDSETPA <Xt1>, <Xt2>, [<Xn|SP>]

### LDSETPAL (A == 1 && R == 1)

LDSETPAL <Xt1>, <Xt2>, [<Xn|SP>]

### LDSETPL (A == 0 && R == 1)

LDSETPL <Xt1>, <Xt2>, [<Xn|SP>]

```
if !HaveLSE128() then UNDEFINED;
if Rt == '11111' then UNDEFINED;
if Rt2 == '11111' then UNDEFINED;
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer n = UInt(Rn);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable LSE128OVERLAP);
  assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
  case c of
    when Constraint UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
    when Constraint UNDEF   UNDEFINED;
    when Constraint NOP     EndOfInstruction();
```

## Assembler Symbols

- <Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(64) value1 = X[t, 64];
bits(64) value2 = X[t2, 64];
bits(128) data;
bits(128) store_value;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

store_value = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

bits(128) comparevalue = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

if rt_unknown then
    data = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = data<127:64>;
    X[t2, 64] = data<63:0>;
else
    X[t, 64] = data<63:0>;
    X[t2, 64] = data<127:64>;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL

Atomic signed maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSMAXA and LDSMAXAL load from memory with acquire semantics.
- LDSMAXL and LDSMAXAL store to memory with release semantics.
- LDSMAX has neither acquire nor release semantics.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias *STSMAX, STSMAXL*.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	A	R	1	Rs				0	1	0	0	0	0	Rn				Rt						
size																opc															

32-bit LDSMAX (size == 10 && A == 0 && R == 0)

```
LDSMAX <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMAXA (size == 10 && A == 1 && R == 0)

```
LDSMAXA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMAXAL (size == 10 && A == 1 && R == 1)

```
LDSMAXAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMAXL (size == 10 && A == 0 && R == 1)

```
LDSMAXL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDSMAX (size == 11 && A == 0 && R == 0)

```
LDSMAX <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMAXA (size == 11 && A == 1 && R == 0)

```
LDSMAXA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMAXAL (size == 11 && A == 1 && R == 1)

```
LDSMAXAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMAXL (size == 11 && A == 0 && R == 1)

```
LDSMAXL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STSMAX</a> , <a href="#">STSMAXL</a>	A == '0' && Rt == '11111'



## Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMAX, acquire, release, tagchecked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(datasize) comparevalue = bits(datasize) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB

Atomic signed maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAB and LDSMAXALB load from memory with acquire semantics.
- LDSMAXLB and LDSMAXALB store to memory with release semantics.
- LDSMAXB has neither acquire nor release semantics.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias [STSMAXB, STSMAXLB](#).

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	A	R	1	Rs				0	1	0	0	0	0	Rn				Rt						
size												opc																			

LDSMAXAB (A == 1 && R == 0)

LDSMAXAB <Ws>, <Wt>, [<Xn|SP>]

LDSMAXALB (A == 1 && R == 1)

LDSMAXALB <Ws>, <Wt>, [<Xn|SP>]

LDSMAXB (A == 0 && R == 0)

LDSMAXB <Ws>, <Wt>, [<Xn|SP>]

LDSMAXLB (A == 0 && R == 1)

LDSMAXLB <Ws>, <Wt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STSMAXB, STSMAXLB</a>	A == '0' && Rt == '11111'

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMAX, acquire, release, tagchecked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH

Atomic signed maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

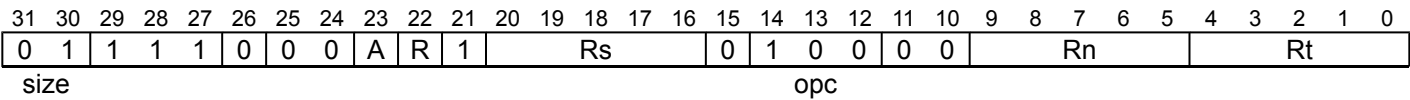
- If the destination register is not WZR, LDSMAXAH and LDSMAXALH load from memory with acquire semantics.
- LDSMAXLH and LDSMAXALH store to memory with release semantics.
- LDSMAXH has neither acquire nor release semantics.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias [STSMAXH, STSMAXLH](#).

Integer  
(FEAT\_LSE)



LDSMAXAH (A == 1 && R == 0)

```
LDSMAXAH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXALH (A == 1 && R == 1)

```
LDSMAXALH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXH (A == 0 && R == 0)

```
LDSMAXH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXLH (A == 0 && R == 1)

```
LDSMAXLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STSMAXH, STSMAXLH</a>	A == '0' && Rt == '11111'

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMAX, acquire, release, tagchecked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDSMIN, LDSMINA, LDSMINAL, LDSMINL

Atomic signed minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

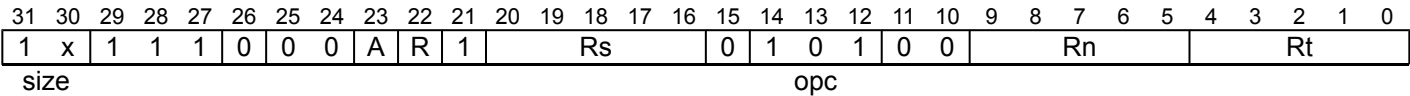
- If the destination register is not one of WZR or XZR, LDSMINA and LDSMINAL load from memory with acquire semantics.
- LDSMINL and LDSMINAL store to memory with release semantics.
- LDSMIN has neither acquire nor release semantics.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias *STSMIN, STSMINL*.

Integer  
(FEAT\_LSE)



32-bit LDSMIN (size == 10 && A == 0 && R == 0)

```
LDSMIN <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMINA (size == 10 && A == 1 && R == 0)

```
LDSMINA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMINAL (size == 10 && A == 1 && R == 1)

```
LDSMINAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMINL (size == 10 && A == 0 && R == 1)

```
LDSMINL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDSMIN (size == 11 && A == 0 && R == 0)

```
LDSMIN <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMINA (size == 11 && A == 1 && R == 0)

```
LDSMINA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMINAL (size == 11 && A == 1 && R == 1)

```
LDSMINAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMINL (size == 11 && A == 0 && R == 1)

```
LDSMINL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STSMIN</a> , <a href="#">STSMINL</a>	A == '0' && Rt == '11111'

## Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMIN, acquire, release, tagchecked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(datasize) comparevalue = bits(datasize) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB

Atomic signed minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

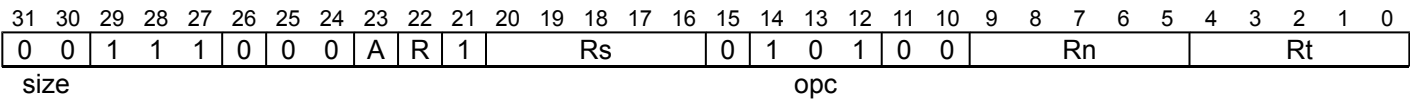
- If the destination register is not WZR, LDSMINAB and LDSMINALB load from memory with acquire semantics.
- LDSMINLB and LDSMINALB store to memory with release semantics.
- LDSMINB has neither acquire nor release semantics.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias *STSMINB, STSMINLB*.

Integer  
(FEAT\_LSE)



LDSMINAB (A == 1 && R == 0)

```
LDSMINAB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINALB (A == 1 && R == 1)

```
LDSMINALB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINB (A == 0 && R == 0)

```
LDSMINB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINLB (A == 0 && R == 1)

```
LDSMINLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STSMINB, STSMINLB</a>	A == '0' && Rt == '11111'

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMIN, acquire, release, tagchecked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH

Atomic signed minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

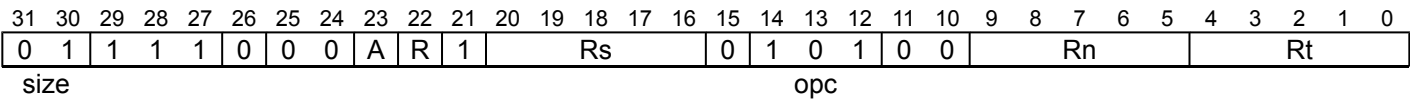
- If the destination register is not WZR, LDSMINAH and LDSMINALH load from memory with acquire semantics.
- LDSMINLH and LDSMINALH store to memory with release semantics.
- LDSMINH has neither acquire nor release semantics.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias [STSMINH, STSMINLH](#).

Integer  
(FEAT\_LSE)



LDSMINAH (A == 1 && R == 0)

```
LDSMINAH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINALH (A == 1 && R == 1)

```
LDSMINALH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINH (A == 0 && R == 0)

```
LDSMINH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINLH (A == 0 && R == 1)

```
LDSMINLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STSMINH, STSMINLH</a>	A == '0' && Rt == '11111'

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMIN, acquire, release, tagchecked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(16) comparevalue = bits(16) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

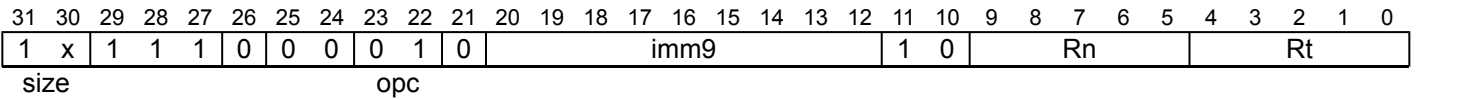
LDTR

Load Register (unprivileged) loads a word or doubleword from memory, and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR\_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



32-bit (size == 10)

```
LDTR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (size == 11)

```
LDTR <Xt>, [<Xn|SP>{, #<simm>}]
```

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tagchecked = n != 31;
```

## Operation

```
bits(64) address;
bits(datasize) data;

boolean privileged = AArch64.IsUnprivAccessPriv\(\);
AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDTRB

Load Register Byte (unprivileged) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR\_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	1	0	imm9										1	0	Rn					Rt			
size											opc																				

LDTRB <Wt>, [<Xn|SP>{, #<sim>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt>
- Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>
- Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim>
- Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(8) data;

boolean privileged = AArch64.IsUnprivAccessPriv();
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

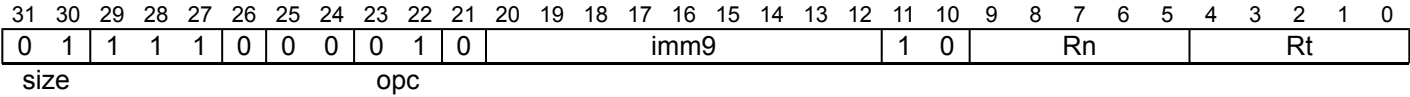
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

LDTRH

Load Register Halfword (unprivileged) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR\_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



LDTRH <Wt>, [<Xn|SP>{, #<sim>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(16) data;

boolean privileged = AArch64.IsUnprivAccessPriv();
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



LDTRSB

Load Register Signed Byte (unprivileged) loads a byte from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR\_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	1	x	0	imm9									1	0	Rn				Rt					
size										opc																					

32-bit (opc == 11)

```
LDTRSB <Wt>, [<Xn|SP>{, #<sim>}]
```

64-bit (opc == 10)

```
LDTRSB <Xt>, [<Xn|SP>{, #<sim>}]
```

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp_PREFETCH && (n != 31);
```

## Operation

```
bits(64) address;
bits(8) data;

boolean privileged = AArch64.IsUnprivAccessPriv\(\);
AccessDescriptor accdesc = CreateAccDescGPR(memop, FALSE, privileged, tagchecked);

if n == 31 then
    if memop != MemOp\_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp\_STORE
        data = X[t, 8];
        Mem[address, 1, accdesc] = data;

    when MemOp\_LOAD
        data = Mem[address, 1, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp\_PREFETCH
        Prefetch(address, t<4:0>);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDTRSH

Load Register Signed Halfword (unprivileged) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR\_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	1	x	0	imm9									1	0	Rn					Rt				
size										opc																					

32-bit (opc == 11)

```
LDTRSH <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (opc == 10)

```
LDTRSH <Xt>, [<Xn|SP>{, #<simm>}]
```

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp_PREFETCH && (n != 31);
```

## Operation

```
bits(64) address;
bits(16) data;

boolean privileged = AArch64.IsUnprivAccessPriv\(\);
AccessDescriptor accdesc = CreateAccDescGPR(memop, FALSE, privileged, tagchecked);

if n == 31 then
    if memop != MemOp\_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp\_STORE
        data = X[t, 16];
        Mem[address, 2, accdesc] = data;

    when MemOp\_LOAD
        data = Mem[address, 2, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp\_PREFETCH
        Prefetch(address, t<4:0>);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDTRSW

Load Register Signed Word (unprivileged) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR\_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	0	0	1	0	0	imm9									1	0	Rn				Rt					
size											opc																				

```
LDTRSW <Xt>, [<Xn|SP>{, #<sim>}]

bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(32) data;

boolean privileged = AArch64.IsUnprivAccessPriv();
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 4, accdesc];
X[t, 64] = SignExtend(data, 64);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL

Atomic unsigned maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDUMAXA and LDUMAXAL load from memory with acquire semantics.
- LDUMAXL and LDUMAXAL store to memory with release semantics.
- LDUMAX has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about memory accesses, see Load/Store addressing modes.

This instruction is used by the alias STUMAX, STUMAXL.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	A	R	1	Rs				0	1	1	0	0	0	Rn				Rt						
size												opc																			

32-bit LDUMAX (size == 10 && A == 0 && R == 0)

```
LDUMAX <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMAXA (size == 10 && A == 1 && R == 0)

```
LDUMAXA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMAXAL (size == 10 && A == 1 && R == 1)

```
LDUMAXAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMAXL (size == 10 && A == 0 && R == 1)

```
LDUMAXL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDUMAX (size == 11 && A == 0 && R == 0)

```
LDUMAX <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMAXA (size == 11 && A == 1 && R == 0)

```
LDUMAXA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMAXAL (size == 11 && A == 1 && R == 1)

```
LDUMAXAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMAXL (size == 11 && A == 0 && R == 1)

```
LDUMAXL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STUMAX, STUMAXL</a>	A == '0' && Rt == '11111'

## Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMAX, acquire, release, tagchecked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(datasize) comparevalue = bits(datasize) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



LDUMAXB, LDUMAXB, LDUMAXALB, LDUMAXLB

Atomic unsigned maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

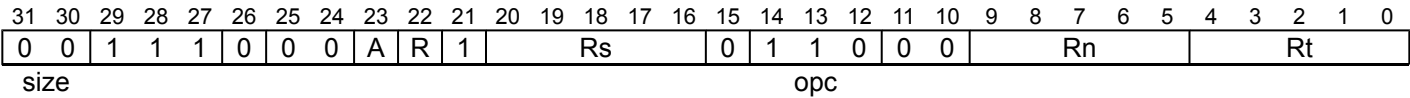
- If the destination register is not WZR, LDUMAXB and LDUMAXALB load from memory with acquire semantics.
- LDUMAXLB and LDUMAXALB store to memory with release semantics.
- LDUMAXB has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about memory accesses, see Load/Store addressing modes.

This instruction is used by the alias STUMAXB, STUMAXLB.

Integer
(FEAT\_LSE)



LDUMAXB (A == 1 && R == 0)

```
LDUMAXB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXALB (A == 1 && R == 1)

```
LDUMAXALB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXB (A == 0 && R == 0)

```
LDUMAXB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXLB (A == 0 && R == 1)

```
LDUMAXLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STUMAXB, STUMAXLB	A == '0' && Rt == '11111'

## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMAX, acquire, release, tagchecked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH

Atomic unsigned maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

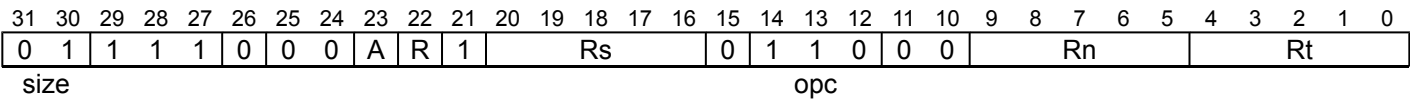
- If the destination register is not WZR, LDUMAXAH and LDUMAXALH load from memory with acquire semantics.
- LDUMAXLH and LDUMAXALH store to memory with release semantics.
- LDUMAXH has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about memory accesses, see Load/Store addressing modes.

This instruction is used by the alias STUMAXH, STUMAXLH.

Integer  
(FEAT\_LSE)



LDUMAXAH (A == 1 && R == 0)

```
LDUMAXAH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXALH (A == 1 && R == 1)

```
LDUMAXALH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXH (A == 0 && R == 0)

```
LDUMAXH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXLH (A == 0 && R == 1)

```
LDUMAXLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STUMAXH, STUMAXLH	A == '0' && Rt == '11111'

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMAX, acquire, release, tagchecked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(16) comparevalue = bits(16) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDUMIN, LDUMINA, LDUMINAL, LDUMINL

Atomic unsigned minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDUMINA and LDUMINAL load from memory with acquire semantics.
- LDUMINL and LDUMINAL store to memory with release semantics.
- LDUMIN has neither acquire nor release semantics.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias *STUMIN, STUMINL*.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	A	R	1	Rs				0	1	1	1	0	0	Rn				Rt						
size												opc																			

32-bit LDUMIN (size == 10 && A == 0 && R == 0)

```
LDUMIN <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMINA (size == 10 && A == 1 && R == 0)

```
LDUMINA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMINAL (size == 10 && A == 1 && R == 1)

```
LDUMINAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMINL (size == 10 && A == 0 && R == 1)

```
LDUMINL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDUMIN (size == 11 && A == 0 && R == 0)

```
LDUMIN <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMINA (size == 11 && A == 1 && R == 0)

```
LDUMINA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMINAL (size == 11 && A == 1 && R == 1)

```
LDUMINAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMINL (size == 11 && A == 0 && R == 1)

```
LDUMINL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STUMIN</a> , <a href="#">STUMINL</a>	A == '0' && Rt == '11111'

## Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMIN, acquire, release, tagchecked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(datasize) comparevalue = bits(datasize) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB

Atomic unsigned minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

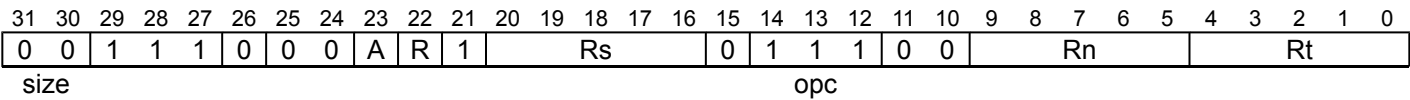
- If the destination register is not WZR, LDUMINAB and LDUMINALB load from memory with acquire semantics.
- LDUMINLB and LDUMINALB store to memory with release semantics.
- LDUMINB has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about memory accesses, see Load/Store addressing modes.

This instruction is used by the alias STUMINB, STUMINLB.

Integer
(FEAT\_LSE)



LDUMINAB (A == 1 && R == 0)

```
LDUMINAB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINALB (A == 1 && R == 1)

```
LDUMINALB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINB (A == 0 && R == 0)

```
LDUMINB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINLB (A == 0 && R == 1)

```
LDUMINLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STUMINB, STUMINLB	A == '0' && Rt == '11111'



## Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMIN, acquire, release, tagchecked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH

Atomic unsigned minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMINAH and LDUMINALH load from memory with acquire semantics.
- LDUMINLH and LDUMINALH store to memory with release semantics.
- LDUMINH has neither acquire nor release semantics.

For more information about memory ordering semantics, see [Load-Acquire, Store-Release](#).

For information about memory accesses, see [Load/Store addressing modes](#).

This instruction is used by the alias [STUMINH, STUMINLH](#).

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs				0	1	1	1	0	0	Rn				Rt						
size											opc																				

LDUMINAH (A == 1 && R == 0)

LDUMINAH <Ws>, <Wt>, [<Xn|SP>]

LDUMINALH (A == 1 && R == 1)

LDUMINALH <Ws>, <Wt>, [<Xn|SP>]

LDUMINH (A == 0 && R == 0)

LDUMINH <Ws>, <Wt>, [<Xn|SP>]

LDUMINLH (A == 0 && R == 1)

LDUMINLH <Ws>, <Wt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
<a href="#">STUMINH, STUMINLH</a>	A == '0' && Rt == '11111'

## Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMIN, acquire, release, tagchecked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

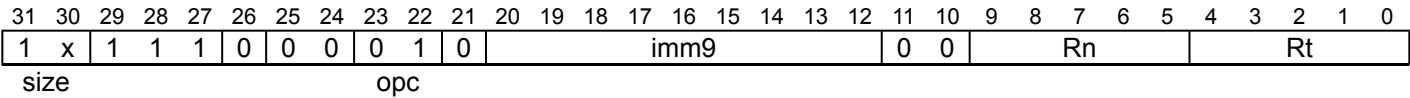
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDUR

Load Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



## 32-bit (size == 10)

```
LDUR <Wt>, [<Xn|SP>{, #<sim>}]
```

## 64-bit (size == 11)

```
LDUR <Xt>, [<Xn|SP>{, #<sim>}]
```

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tagchecked = n != 31;
```

## Operation

```
bits(64) address;
bits(datasize) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

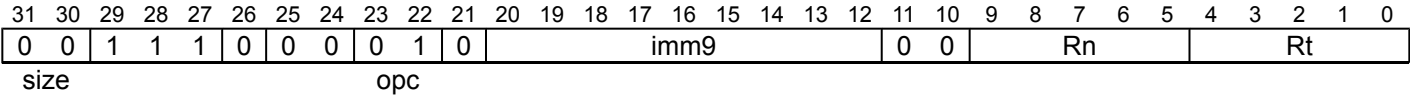
## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



# LDURB

Load Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



LDURB <Wt>, [[<Xn|SP>](#){, #<simm>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

## Operation

```
bits(64) address;
bits(8) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

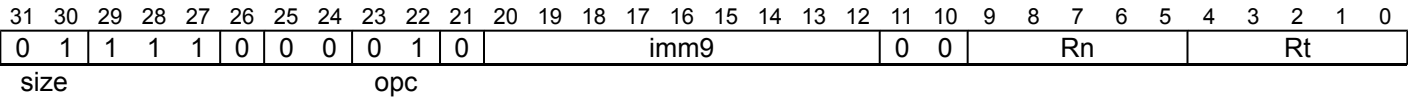
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDURH

Load Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



LDURH <Wt>, [[Xn|SP](#)]{, #<simm>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

## Operation

```
bits(64) address;
bits(16) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

## Operational information

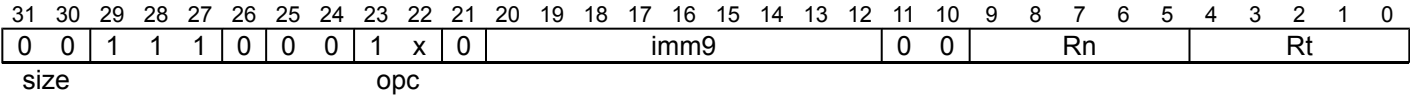
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDURSB

Load Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



## 32-bit (opc == 11)

```
LDURSB <Wt>, [<Xn|SP>{, #<simm>}]
```

## 64-bit (opc == 10)

```
LDURSB <Xt>, [<Xn|SP>{, #<simm>}]
```

```
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp\_LOAD else MemOp\_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp\_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp\_PREFETCH && (n != 31);
```



## Operation

```
bits(64) address;
bits(8) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(memop, FALSE, privileged, tagchecked);

if n == 31 then
    if memop != MemOp\_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp\_STORE
        data = X[t, 8];
        Mem[address, 1, accdesc] = data;

    when MemOp\_LOAD
        data = Mem[address, 1, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp\_PREFETCH
        Prefetch(address, t<4:0>);
```

## Operational information

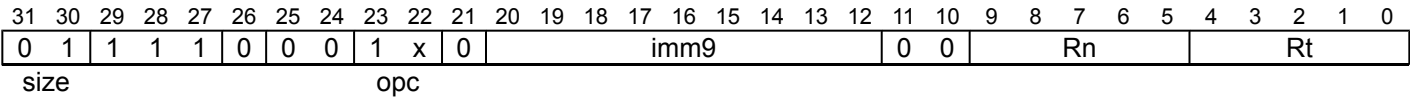
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDURSH

Load Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



## 32-bit (opc == 11)

```
LDURSH <Wt>, [<Xn|SP>{, #<simm>}]
```

## 64-bit (opc == 10)

```
LDURSH <Xt>, [<Xn|SP>{, #<simm>}]
```

```
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp\_LOAD else MemOp\_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp\_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp\_PREFETCH && (n != 31);
```

## Operation

```
bits(64) address;
bits(16) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(memop, FALSE, privileged, tagchecked);

if n == 31 then
    if memop != MemOp\_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp\_STORE
        data = X[t, 16];
        Mem[address, 2, accdesc] = data;

    when MemOp\_LOAD
        data = Mem[address, 2, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp\_PREFETCH
        Prefetch(address, t<4:0>);
```

## Operational information

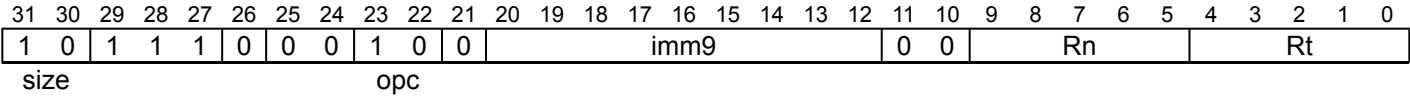
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDURSW

Load Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. For information about memory accesses, see [Load/Store addressing modes](#).



LDURSW <Xt>, [<Xn|SP>{, #<simm>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

## Operation

```
bits(64) address;
bits(32) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 4, accdesc];
X[t, 64] = SignExtend(data, 64);
```

## Operational information

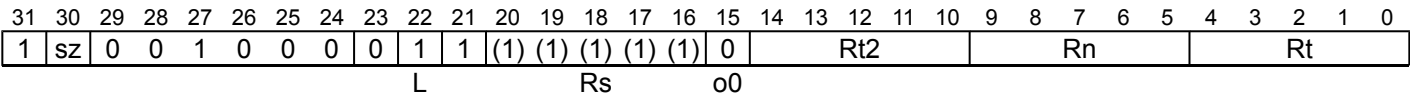
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDXP

Load Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information on single-copy atomicity and alignment requirements, see *Requirements for single-copy atomicity* and *Alignment of data accesses*. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses, see *Load/Store addressing modes*.



## 32-bit (sz == 0)

```
LDXP <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

## 64-bit (sz == 1)

```
LDXP <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDXP*.

## Assembler Symbols

- <Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t, datasize] = bits(datasize) UNKNOWN;    // In this case t = t2
elseif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, accdesc];
    if BigEndian(accdesc.acctype) then
        X[t, datasize-elsize] = data<datasize-1:elsize>;
        X[t2, elsize] = data<elsize-1:0>;
    else
        X[t, elsize] = data<elsize-1:0>;
        X[t2, datasize-elsize] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic), but must be 128-bit aligned
    if !IsAligned(address, dbytes) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    X[t, 64] = Mem[address, 8, accdesc];
    X[t2, 64] = Mem[address+8, 8, accdesc];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDXR

Load Exclusive Register derives an address from a base register value, loads a 32-bit word or a 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	0	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn						Rt			
size									L		Rs				o0		Rt2														

32-bit (size == 10)

```
LDXR <Wt>, [<Xn|SP>{, #0}]
```

64-bit (size == 11)

```
LDXR <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

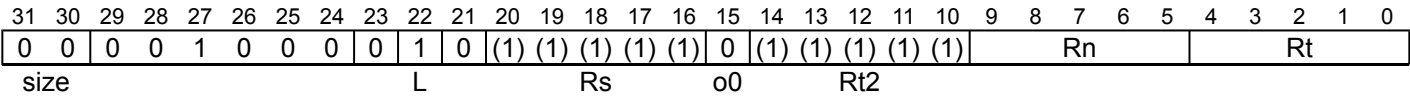
data = Mem[address, dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

LDXRB

Load Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). For information about memory accesses, see [Load/Store addressing modes](#).



```
LDXRB <Wt>, [<Xn|SP>{, #0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

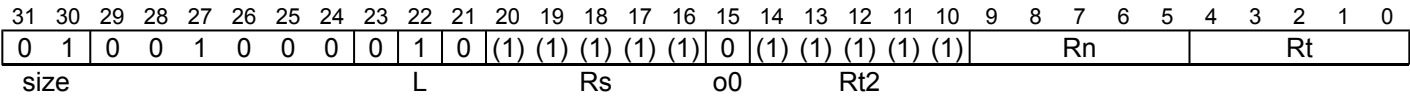
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



LDXRH

Load Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See [Synchronization and semaphores](#). For information about memory accesses, see [Load/Store addressing modes](#).



```
LDXRH <Wt>, [<Xn|SP>{, #0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of [UBFM](#). This means:

- The encodings in this description are named to match the encodings of [UBFM](#).
- The description of [UBFM](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0												
sf		1		0		1		0		0		1		1		0		N		immr						!= x11111						Rn						Rd					
opc																						imms																					

32-bit (sf == 0 && N == 0 && imms != 011111)

LSL <Wd>, <Wn>, #<shift>

is equivalent to

UBFM <Wd>, <Wn>, #(-<shift> MOD 32), #(31-<shift>)

and is the preferred disassembly when `imms + 1 == immr`.

64-bit (sf == 1 && N == 1 && imms != 111111)

LSL <Xd>, <Xn>, #<shift>

is equivalent to

UBFM <Xd>, <Xn>, #(-<shift> MOD 64), #(63-<shift>)

and is the preferred disassembly when `imms + 1 == immr`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<shift>	For the 32-bit variant: is the shift amount, in the range 0 to 31. For the 64-bit variant: is the shift amount, in the range 0 to 63.

Operation

The description of [UBFM](#) gives the operational pseudocode for this instruction.

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

LSL (register)

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This is an alias of [LSLV](#). This means:

- The encodings in this description are named to match the encodings of [LSLV](#).
- The description of [LSLV](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	0	0	0	Rn					Rd				

op2

32-bit (sf == 0)

LSL <Wd>, <Wn>, <Wm>

is equivalent to

[LSLV](#) <Wd>, <Wn>, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

LSL <Xd>, <Xn>, <Xm>

is equivalent to

[LSLV](#) <Xd>, <Xn>, <Xm>

and is always the preferred disassembly.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of [LSLV](#) gives the operational pseudocode for this instruction.

Operational information

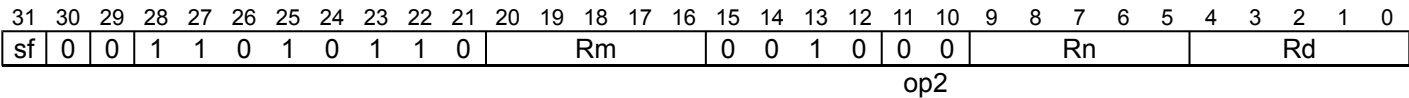
- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



LSLV

Logical Shift Left Variable shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This instruction is used by the alias [LSL \(register\)](#).



32-bit (sf == 0)

```
LSLV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
LSLV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m, datasize];

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);
X[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of [UBFM](#). This means:

- The encodings in this description are named to match the encodings of [UBFM](#).
- The description of [UBFM](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	1	0	0	1	1	0	N	immr						x	1	1	1	1	1	Rn					Rd				
opc										imms																					

32-bit (sf == 0 && N == 0 && imms == 011111)

LSR <Wd>, <Wn>, #<shift>

is equivalent to

UBFM <Wd>, <Wn>, #<shift>, #31

and is always the preferred disassembly.

64-bit (sf == 1 && N == 1 && imms == 111111)

LSR <Xd>, <Xn>, #<shift>

is equivalent to

UBFM <Xd>, <Xn>, #<shift>, #63

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<shift>	For the 32-bit variant: is the shift amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, encoded in the "immr" field.

Operation

The description of [UBFM](#) gives the operational pseudocode for this instruction.

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

## LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This is an alias of [LSRV](#). This means:

- The encodings in this description are named to match the encodings of [LSRV](#).
- The description of [LSRV](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	0	0	1	Rn					Rd				

op2

### 32-bit (sf == 0)

LSR <Wd>, <Wn>, <Wm>

is equivalent to

[LSRV](#) <Wd>, <Wn>, <Wm>

and is always the preferred disassembly.

### 64-bit (sf == 1)

LSR <Xd>, <Xn>, <Xm>

is equivalent to

[LSRV](#) <Xd>, <Xn>, <Xm>

and is always the preferred disassembly.

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

## Operation

The description of [LSRV](#) gives the operational pseudocode for this instruction.

## Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

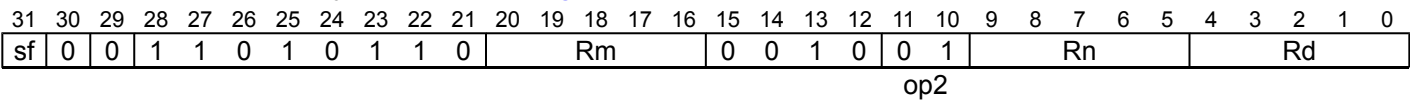




LSRV

Logical Shift Right Variable shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias [LSR \(register\)](#).



32-bit (sf == 0)

```
LSRV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
LSRV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m, datasize];

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);
X[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

MADD

Multiply-Add multiplies two register values, adds a third register value, and writes the result to the destination register.

This instruction is used by the alias [MUL](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	1	0	0	0	Rm				0	Ra				Rn				Rd							

o0

32-bit (sf == 0)

MADD <Wd>, <Wn>, <Wm>, <Wa>

64-bit (sf == 1)

MADD <Xd>, <Xn>, <Xm>, <Xa>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32;
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- <Wa> Is the 32-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- <Xa> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
<a href="#">MUL</a>	Ra == '11111'

Operation

```
bits(destsize) operand1 = X[n, destsize];
bits(destsize) operand2 = X[m, destsize];
bits(destsize) operand3 = X[a, destsize];

integer result;

result = UInt(operand3) + (UInt(operand1) * UInt(operand2));
X[d, destsize] = result<destsize-1:0>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# MNEG

Multiply-Negate multiplies two register values, negates the product, and writes the result to the destination register.

This is an alias of [MSUB](#). This means:

- The encodings in this description are named to match the encodings of [MSUB](#).
- The description of [MSUB](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																				
sf		0		0		1		1		0		1		1		0		0		0		Rm						1		1		1		1		1		1		Rn						Rd					
																o0		Ra																																	

## 32-bit (sf == 0)

MNEG [<Wd>](#), [<Wn>](#), [<Wm>](#)

is equivalent to

[MSUB](#) [<Wd>](#), [<Wn>](#), [<Wm>](#), WZR

and is always the preferred disassembly.

## 64-bit (sf == 1)

MNEG [<Xd>](#), [<Xn>](#), [<Xm>](#)

is equivalent to

[MSUB](#) [<Xd>](#), [<Xn>](#), [<Xm>](#), XZR

and is always the preferred disassembly.

## Assembler Symbols

- [<Wd>](#) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- [<Wn>](#) Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- [<Wm>](#) Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- [<Xd>](#) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- [<Xn>](#) Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- [<Xm>](#) Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

## Operation

The description of [MSUB](#) gives the operational pseudocode for this instruction.

## Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



MOV (bitmask immediate)

Move (bitmask immediate) writes a bitmask immediate value to a register.

This is an alias of [ORR \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [ORR \(immediate\)](#).
- The description of [ORR \(immediate\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
sf	0	1	1	0	0	1	0	0	N	immr						imms						1	1	1	1	1	Rd					
opc																Rn																

32-bit (sf == 0 && N == 0)

```
MOV <Wd|WSP>, #<imm>
```

is equivalent to

```
ORR <Wd|WSP>, WZR, #<imm>
```

and is the preferred disassembly when ! MoveWidePreferred(sf, N, imms, immr).

64-bit (sf == 1)

```
MOV <Xd|SP>, #<imm>
```

is equivalent to

```
ORR <Xd|SP>, XZR, #<imm>
```

and is the preferred disassembly when ! MoveWidePreferred(sf, N, imms, immr).

Assembler Symbols

- <Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <imm> For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr", but excluding values which could be encoded by MOVZ or MOVN.  
For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr", but excluding values which could be encoded by MOVZ or MOVN.

Operation

The description of [ORR \(immediate\)](#) gives the operational pseudocode for this instruction.

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

# MOV (inverted wide immediate)

Move (inverted wide immediate) moves an inverted 16-bit immediate value to a register.

This is an alias of [MOVN](#). This means:

- The encodings in this description are named to match the encodings of [MOVN](#).
- The description of [MOVN](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		0	0	1	0	0	1	0	1	hw		imm16																Rd			
opc																															

## 32-bit (sf == 0 && hw == 0x)

MOV <Wd>, #<imm>

is equivalent to

MOVN <Wd>, #<imm16>, LSL #<shift>

and is the preferred disassembly when `!(IsZero(imm16) && hw != '00') && ! IsOnes(imm16)`.

## 64-bit (sf == 1)

MOV <Xd>, #<imm>

is equivalent to

MOVN <Xd>, #<imm16>, LSL #<shift>

and is the preferred disassembly when `!(IsZero(imm16) && hw != '00')`.

## Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <imm> For the 32-bit variant: is a 32-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw", but excluding 0xffff0000 and 0x0000ffff  
For the 64-bit variant: is a 64-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw".
- <shift> For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.  
For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

## Operation

The description of [MOVN](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.





# MOV (register)

Move (register) copies the value in a source register to the destination register.

This is an alias of [ORR \(shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [ORR \(shifted register\)](#).
- The description of [ORR \(shifted register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	1	0	1	0	1	0	0	0	0	Rm				0	0	0	0	0	0	1	1	1	1	1	Rd					
opc				shift				N				imm6						Rn													

## 32-bit (sf == 0)

MOV <Wd>, <Wm>

is equivalent to

ORR <Wd>, WZR, <Wm>

and is always the preferred disassembly.

## 64-bit (sf == 1)

MOV <Xd>, <Xm>

is equivalent to

ORR <Xd>, XZR, <Xm>

and is always the preferred disassembly.

## Assembler Symbols

- |      |  |
|------|--|
| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Wm> | Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.      |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xm> | Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.      |

## Operation

The description of [ORR \(shifted register\)](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# MOV (to/from SP)

Move between register and stack pointer

: Rd = Rn.

This is an alias of [ADD \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [ADD \(immediate\)](#).
- The description of [ADD \(immediate\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
sf		0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	Rn						Rd					
op				S								sh				imm12																	

## 32-bit (sf == 0)

MOV <Wd|WSP>, <Wn|WSP>

is equivalent to

ADD <Wd|WSP>, <Wn|WSP>, #0

and is the preferred disassembly when (Rd == '11111' || Rn == '11111').

## 64-bit (sf == 1)

MOV <Xd|SP>, <Xn|SP>

is equivalent to

ADD <Xd|SP>, <Xn|SP>, #0

and is the preferred disassembly when (Rd == '11111' || Rn == '11111').

## Assembler Symbols

- <Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Wn|WSP> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

## Operation

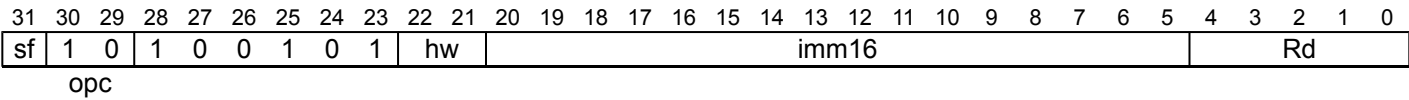
The description of [ADD \(immediate\)](#) gives the operational pseudocode for this instruction.

MOV (wide immediate)

Move (wide immediate) moves a 16-bit immediate value to a register.

This is an alias of MOVZ. This means:

- The encodings in this description are named to match the encodings of MOVZ.
- The description of MOVZ gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit (sf == 0 && hw == 0x)

```
MOV <Wd>, #<imm>
```

is equivalent to

```
MOVZ <Wd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when ! (IsZero(imm16) && hw != '00').

64-bit (sf == 1)

```
MOV <Xd>, #<imm>
```

is equivalent to

```
MOVZ <Xd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when ! (IsZero(imm16) && hw != '00').

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <imm> For the 32-bit variant: is a 32-bit immediate which can be encoded in "imm16:hw".  
For the 64-bit variant: is a 64-bit immediate which can be encoded in "imm16:hw".
- <shift> For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.  
For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

The description of MOVZ gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



MOVK

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
sf		1		1		1		0		0		1		0		1		hw		imm16																Rd			

opc

32-bit (sf == 0 && hw == 0x)

```
MOVK <Wd>, #<imm>{, LSL #<shift>}
```

64-bit (sf == 1)

```
MOVK <Xd>, #<imm>{, LSL #<shift>}
```

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;

if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = UInt(hw:'0000');
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <imm> Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
- <shift> For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.  
For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

```
bits(datasize) result;

result = X[d, datasize];
result<pos+15:pos> = imm16;
X[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

MOVN

Move wide with NOT moves the inverse of an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias [MOV \(inverted wide immediate\)](#).

313029282726252423222120191817161514131211109876543210

sf

00100101

hw

imm16

Rd

opc

32-bit (sf == 0 && hw == 0x)

MOVN <Wd>, #<imm>{, LSL #<shift>}

64-bit (sf == 1)

MOVN <Xd>, #<imm>{, LSL #<shift>}

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;

if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = UInt(hw:'0000');
```

Assembler Symbols

- <Wd>
- Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xd>
- Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <imm>
- Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
- <shift>
- For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.
- For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Alias Conditions

Alias	Of variant	Is preferred when
<a href="#">MOV (inverted wide immediate)</a>	64-bit	! ( <a href="#">IsZero</a> (imm16) && hw != '00')
<a href="#">MOV (inverted wide immediate)</a>	32-bit	! ( <a href="#">IsZero</a> (imm16) && hw != '00') && ! <a href="#">IsOnes</a> (imm16)

Operation

```
bits(datasize) result;

result = Zeros(datasize);

result<pos+15:pos> = imm16;
result = NOT(result);
X[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

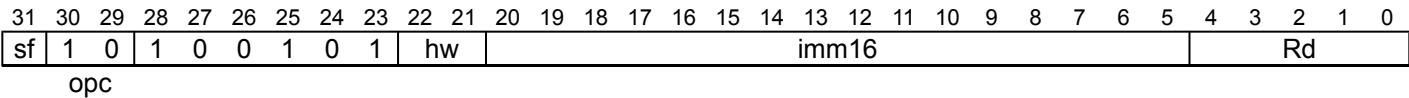
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# MOVZ

Move wide with zero moves an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias [MOV \(wide immediate\)](#).



32-bit (sf == 0 && hw == 0x)

```
MOVZ <Wd>, #<imm>{, LSL #<shift>}
```

64-bit (sf == 1)

```
MOVZ <Xd>, #<imm>{, LSL #<shift>}
```

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;

if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = UInt(hw:'0000');
```

## Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <imm> Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
- <shift> For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.  
For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

## Alias Conditions

Alias	Is preferred when
<a href="#">MOV (wide immediate)</a>	! (IsZero(imm16) && hw != '00')

## Operation

```
bits(datasize) result;

result = Zeros(datasize);

result<pos+15:pos> = imm16;
X[d, datasize] = result;
```

## Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

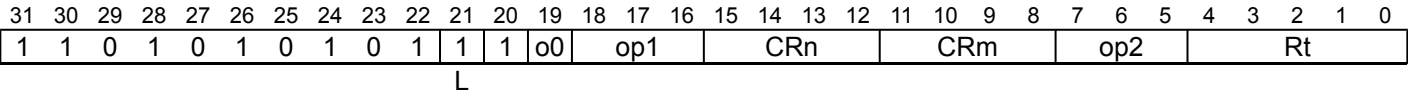




MRRS

Move System Register to two adjacent general-purpose registers allows the PE to read an AArch64 128-bit System register into two adjacent 64-bit general-purpose registers.

System  
(FEAT\_SYSREG128)



MRRS <Xt>, <Xt+1>, (<systemreg>|S<op0>\_<op1>\_<Cn>\_<Cm>\_<op2>)

```
if !HaveSysReg128() then UNDEFINED;
if Rt<0> == '1' then UNDEFINED;
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);
integer t2 = UInt(Rt + 1);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler Symbols

- <Xt> Is the 64-bit name of the first general-purpose destination register, encoded in the "Rt" field.
- <Xt+1> Is the 64-bit name of the second general-purpose destination register, encoded as "Rt" +1.
- <systemreg> Is a System register name, encoded in "o0:op1:CRn:CRm:op2".
- <op0> Is an unsigned immediate, encoded in "o0":

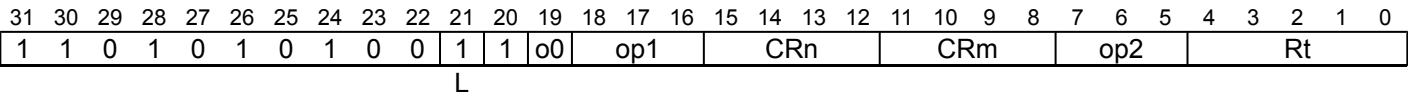
o0	<op0>
0	2
1	3
- <op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- <Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
- <Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- <op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

Operation

```
AArch64.SysRegRead128(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t, t2);
```

MRS

Move System Register to general-purpose register allows the PE to read an AArch64 System register into a general-purpose register.



MRS <Xt>, (<systemreg>|S<op0>\_<op1>\_<Cn>\_<Cm>\_<op2>)

```
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.
- <systemreg> Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".  
The System register names are defined in 'AArch64 System Registers' in the System Register XML.
- <op0> Is an unsigned immediate, encoded in "o0":

o0	<op0>
0	2
1	3
- <op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- <Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
- <Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- <op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

Operation

```
AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
```

MSR (immediate)

Move immediate value to Special Register moves an immediate value to selected bits of the PSTATE. For more information, see [Process state, PSTATE](#).

The bits that can be written by this instruction are:

- PSTATE.D, PSTATE.A, PSTATE.I, PSTATE.F, and PSTATE.SP.
- If [FEAT\\_SSBS](#) is implemented, PSTATE.SSBS.
- If [FEAT\\_PAN](#) is implemented, PSTATE.PAN.
- If [FEAT\\_UAO](#) is implemented, PSTATE.UAO.
- If [FEAT\\_DIT](#) is implemented, PSTATE.DIT.
- If [FEAT\\_MTE](#) is implemented, PSTATE.TCO.
- If [FEAT\\_NMI](#) is implemented, PSTATE.ALLINT.
- If [FEAT\\_SME](#) is implemented, PSTATE.SM and PSTATE.ZA.
- If FEAT\_EBEP is implemented, PSTATE.PM.

This instruction is used by the aliases [SMSTART](#), and [SMSTOP](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	op1			0	1	0	0	CRm			op2			1	1	1	1	1	

MSR <pstatefield>, #<imm>

```

if op1 == '000' && op2 == '000' then SEE "CFINV";
if op1 == '000' && op2 == '001' then SEE "XAFLAG";
if op1 == '000' && op2 == '010' then SEE "AXFLAG";

AArch64.CheckSystemAccess('00', op1, '0100', CRm, op2, '11111', '0');
bits(2) min_EL;
boolean need_secure = FALSE;

case op1 of
  when '00x'
    min_EL = EL1;
  when '010'
    min_EL = EL1;
  when '011'
    min_EL = EL0;
  when '100'
    min_EL = EL2;
  when '101'
    if !HaveVirtHostExt() then
      UNDEFINED;
    min_EL = EL2;
  when '110'
    min_EL = EL3;
  when '111'
    min_EL = EL1;
    need_secure = TRUE;

if (UInt(PSTATE.EL) < UInt(min_EL) || (need_secure && CurrentSecurityState() != SS\_Secure)) then
  UNDEFINED;

PSTATEField field;
case op1:op2 of
  when '000 011'
    if !HaveUAOExt() then UNDEFINED;
    field = PSTATEField\_UAO;
  when '000 100'
    if !HavePANExt() then UNDEFINED;
    field = PSTATEField\_PAN;
  when '000 101' field = PSTATEField\_SP;
  when '001 000'
    case CRm of
      when '000x'
        if !HaveFeatNMI() then UNDEFINED;
        field = PSTATEField\_ALLINT;
      when '001x'
        if !IsFeatureImplemented(FEAT_EBEP) then UNDEFINED;
        field = PSTATEField\_PM;
      otherwise
        UNDEFINED;
  when '011 010'
    if !HaveDITEExt() then UNDEFINED;
    field = PSTATEField\_DIT;
  when '011 011'
    case CRm of
      when '001x'
        if !HaveSME() then UNDEFINED;
        field = PSTATEField\_SVCRSM;
      when '010x'
        if !HaveSME() then UNDEFINED;
        field = PSTATEField\_SVCRZA;
      when '011x'
        if !HaveSME() then UNDEFINED;
        field = PSTATEField\_SVCRSMZA;
      otherwise
        UNDEFINED;
  when '011 100'
    if !HaveMTEExt() then UNDEFINED;
    field = PSTATEField\_TCO;
  when '011 110' field = PSTATEField\_DAIFFSet;
  when '011 111' field = PSTATEField\_DAIFFClr;

```

```

when '011 001'
    if !HaveSSBSExt() then UNDEFINED;
    field = PSTATEField SSBS;
otherwise UNDEFINED;

```

## Assembler Symbols

<pstatefield> Is a PSTATE field name. For the MSR instruction, this is encoded in “op1:op2:CRm”:

op1	op2	CRm	<pstatefield>	Architectural Feature
000	00x	xxxx	<a href="#">SEE PSTATE</a>	-
000	010	xxxx	<a href="#">SEE PSTATE</a>	-
000	011	xxxx	UAO	FEAT_UAO
000	100	xxxx	PAN	FEAT_PAN
000	101	xxxx	SPSel	-
000	11x	xxxx	RESERVED	-
001	000	000x	ALLINT	FEAT_NMI
001	000	001x	PM	FEAT_EBEP
001	000	01xx	RESERVED	-
001	000	1xxx	RESERVED	-
001	001	xxxx	RESERVED	-
001	01x	xxxx	RESERVED	-
001	1xx	xxxx	RESERVED	-
010	xxx	xxxx	RESERVED	-
011	000	xxxx	RESERVED	-
011	001	xxxx	SSBS	FEAT_SSBS
011	010	xxxx	DIT	FEAT_DIT
011	011	000x	RESERVED	-
011	011	001x	SVCRSM	FEAT_SME
011	011	010x	SVCRZA	FEAT_SME
011	011	011x	SVCRSMZA	FEAT_SME
011	011	1xxx	RESERVED	-
011	100	xxxx	TCO	FEAT_MTE
011	101	xxxx	RESERVED	-
011	110	xxxx	DAIFSet	-
011	111	xxxx	DAIFClr	-
1xx	xxx	xxxx	RESERVED	-

For the SMSTART and SMSTOP aliases, this is encoded in "CRm<2:1>", where 0b01 specifies SVCRSM, 0b10 specifies SVCRZA, and 0b11 specifies SVCRSMZA.

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field. Restricted to the range 0 to 1, encoded in "CRm<0>", when <pstatefield> is ALLINT, PM, SVCRSM, SVCRSMZA, or SVCRZA.

## Alias Conditions

Alias	Is preferred when
<a href="#">SMSTART</a>	op1 == '011' && CRm == '0xx1' && op2 == '011'
<a href="#">SMSTOP</a>	op1 == '011' && CRm == '0xx0' && op2 == '011'

## Operation

```
case field of
  when PSTATEField\_SSBS
    PSTATE.SSBS = CRm<0>;
  when PSTATEField\_SP
    PSTATE.SP = CRm<0>;
  when PSTATEField\_DAIFFSet
    AArch64.CheckDAIFAccess(PSTATEField\_DAIFFSet);
    PSTATE.D = PSTATE.D OR CRm<3>;
    PSTATE.A = PSTATE.A OR CRm<2>;
    PSTATE.I = PSTATE.I OR CRm<1>;
    PSTATE.F = PSTATE.F OR CRm<0>;
  when PSTATEField\_DAIFFClr
    AArch64.CheckDAIFAccess(PSTATEField\_DAIFFClr);
    PSTATE.D = PSTATE.D AND NOT(CRm<3>);
    PSTATE.A = PSTATE.A AND NOT(CRm<2>);
    PSTATE.I = PSTATE.I AND NOT(CRm<1>);
    PSTATE.F = PSTATE.F AND NOT(CRm<0>);
  when PSTATEField\_PAN
    PSTATE.PAN = CRm<0>;
  when PSTATEField\_UAO
    PSTATE.UAO = CRm<0>;
  when PSTATEField\_DIT
    PSTATE.DIT = CRm<0>;
  when PSTATEField\_TCO
    PSTATE.TCO = CRm<0>;
  when PSTATEField\_ALLINT
    if (PSTATE.EL == EL1 && IsHCRXEL2Enabled() && HCRX_EL2.TALLINT == '1' && CRm<0> == '1') then
      AArch64.SystemAccessTrap(EL2, 0x18);
    PSTATE.ALLINT = CRm<0>;
  when PSTATEField\_SVCRSM
    CheckSMEAccess();
    SetPSTATE\_SM(CRm<0>);
  when PSTATEField\_SVCRZA
    CheckSMEAccess();
    SetPSTATE\_ZA(CRm<0>);
  when PSTATEField\_SVCRSMZA
    CheckSMEAccess();
    SetPSTATE\_SM(CRm<0>);
    SetPSTATE\_ZA(CRm<0>);
  when PSTATEField\_PM
    PSTATE.PM = CRm<0>;
```

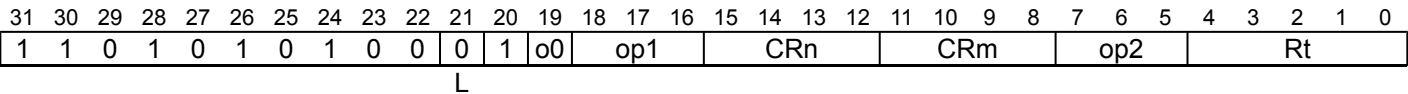
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



MSR (register)

Move general-purpose register to System Register allows the PE to write an AArch64 System register from a general-purpose register.



MSR (<systemreg>|S<op0>\_<op1>\_<Cn>\_<Cm>\_<op2>), <Xt>

```
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler Symbols

- <systemreg> Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".  
The System register names are defined in 'AArch64 System Registers' in the System Register XML.
- <op0> Is an unsigned immediate, encoded in "o0":

o0	<op0>
0	2
1	3
- <op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- <Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
- <Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- <op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
- <Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

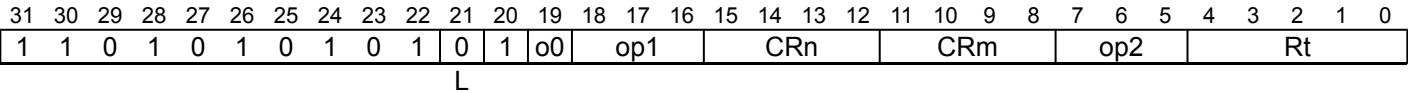
Operation

```
AArch64.SysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
```

MSRR

Move two adjacent general-purpose registers to System Register allows the PE to write an AArch64 128-bit System register from two adjacent 64-bit general-purpose registers.

System  
(FEAT\_SYSREG128)



MSRR (<systemreg>|s<op0>\_<op1>\_<Cn>\_<Cm>\_<op2>), <Xt>, <Xt+1>

```
if !HaveSysReg128() then UNDEFINED;
if Rt<0> == '1' then UNDEFINED;
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);
integer t2 = UInt(Rt + 1);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler Symbols

- <systemreg> Is a System register name, encoded in "o0:op1:CRn:CRm:op2".
- <op0> Is an unsigned immediate, encoded in "o0":

o0		<op0>
0	2	
1	3	
- <op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- <Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
- <Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- <op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
- <Xt> Is the 64-bit name of the first general-purpose source register, encoded in the "Rt" field.
- <Xt+1> Is the 64-bit name of the second general-purpose source register, encoded as "Rt" + 1.

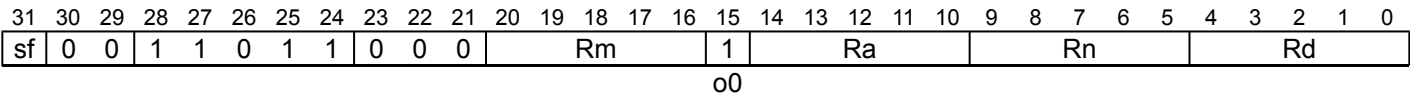
Operation

```
AArch64.SysRegWrite128(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t, t2);
```

MSUB

Multiply-Subtract multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register.

This instruction is used by the alias [MNEG](#).



32-bit (sf == 0)

```
MSUB <Wd>, <Wn>, <Wm>, <Wa>
```

64-bit (sf == 1)

```
MSUB <Xd>, <Xn>, <Xm>, <Xa>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32;
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- <Wa> Is the 32-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- <Xa> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
<a href="#">MNEG</a>	Ra == '11111'

Operation

```
bits(destsize) operand1 = X[n, destsize];
bits(destsize) operand2 = X[m, destsize];
bits(destsize) operand3 = X[a, destsize];

integer result;

result = UInt(operand3) - (UInt(operand1) * UInt(operand2));
X[d, destsize] = result<destsize-1:0>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# MUL

Multiply

: Rd = Rn \* Rm.

This is an alias of [MADD](#). This means:

- The encodings in this description are named to match the encodings of [MADD](#).
- The description of [MADD](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
sf	0	0	1	1	0	1	1	0	0	0	Rm					0	1	1	1	1	1	Rn					Rd					
																o0		Ra														

## 32-bit (sf == 0)

MUL <Wd>, <Wn>, <Wm>

is equivalent to

[MADD](#) <Wd>, <Wn>, <Wm>, WZR

and is always the preferred disassembly.

## 64-bit (sf == 1)

MUL <Xd>, <Xn>, <Xm>

is equivalent to

[MADD](#) <Xd>, <Xn>, <Xm>, XZR

and is always the preferred disassembly.

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

## Operation

The description of [MADD](#) gives the operational pseudocode for this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# MVN

Bitwise NOT writes the bitwise inverse of a register value to the destination register.

This is an alias of [ORN \(shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [ORN \(shifted register\)](#).
- The description of [ORN \(shifted register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		0	1	0	1	0	1	0	shift		1	Rm				imm6				1				1	1	1	1	Rd			
opc										N		Rn																			

## 32-bit (sf == 0)

MVN <Wd>, <Wm>{, <shift> #<amount>}

is equivalent to

ORN <Wd>, WZR, <Wm>{, <shift> #<amount>}

and is always the preferred disassembly.

## 64-bit (sf == 1)

MVN <Xd>, <Xm>{, <shift> #<amount>}

is equivalent to

ORN <Xd>, XZR, <Xm>{, <shift> #<amount>}

and is always the preferred disassembly.

## Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.
- <shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

- <amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
- For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

## Operation

The description of [ORN \(shifted register\)](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## NEG (shifted register)

Negate (shifted register) negates an optionally-shifted register value, and writes the result to the destination register.

This is an alias of [SUB \(shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [SUB \(shifted register\)](#).
- The description of [SUB \(shifted register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
sf	1	0	0	1	0	1	1	shift	0	Rm						imm6						1	1	1	1	1	Rd					
op S												Rn																				

### 32-bit (sf == 0)

NEG <Wd>, <Wm>{, <shift> #<amount>}

is equivalent to

SUB <Wd>, WZR, <Wm> {, <shift> #<amount>}

and is always the preferred disassembly.

### 64-bit (sf == 1)

NEG <Xd>, <Xm>{, <shift> #<amount>}

is equivalent to

SUB <Xd>, XZR, <Xm> {, <shift> #<amount>}

and is always the preferred disassembly.

## Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.
- <shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

- <amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
- For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

## Operation

The description of [SUB \(shifted register\)](#) gives the operational pseudocode for this instruction.



## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

NEGS

Negate, setting flags, negates an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of [SUBS \(shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [SUBS \(shifted register\)](#).
- The description of [SUBS \(shifted register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
sf	1	1	0	1	0	1	1	shift	0	Rm						imm6						1	1	1	1	1	!= 11111						
op S																								Rn					Rd				

32-bit (sf == 0)

NEGS <Wd>, <Wm>{, <shift> #<amount>}

is equivalent to

SUBS <Wd>, WZR, <Wm> {, <shift> #<amount>}

and is always the preferred disassembly.

64-bit (sf == 1)

NEGS <Xd>, <Xm>{, <shift> #<amount>}

is equivalent to

SUBS <Xd>, XZR, <Xm> {, <shift> #<amount>}

and is always the preferred disassembly.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

The description of [SUBS \(shifted register\)](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

NGC

Negate with Carry negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register.

This is an alias of [SBC](#). This means:

- The encodings in this description are named to match the encodings of [SBC](#).
- The description of [SBC](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
sf		1	0	1	1	0	1	0	0	0	0	Rm				0	0	0	0	0	0	1	1	1	1	1	Rd												
op												S																Rn											

32-bit (sf == 0)

NGC [<Wd>](#), [<Wm>](#)

is equivalent to

[SBC](#) [<Wd>](#), WZR, [<Wm>](#)

and is always the preferred disassembly.

64-bit (sf == 1)

NGC [<Xd>](#), [<Xm>](#)

is equivalent to

[SBC](#) [<Xd>](#), XZR, [<Xm>](#)

and is always the preferred disassembly.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of [SBC](#) gives the operational pseudocode for this instruction.

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

NGCS

Negate with Carry, setting flags, negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of [SBCS](#). This means:

- The encodings in this description are named to match the encodings of [SBCS](#).
- The description of [SBCS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
sf		1	1	1	1	0	1	0	0	0	0	Rm				0	0	0	0	0	0	0	1	1	1	1	1	Rd											
op												S																Rn											

32-bit (sf == 0)

NGCS [<Wd>](#), [<Wm>](#)

is equivalent to

[SBCS](#) [<Wd>](#), WZR, [<Wm>](#)

and is always the preferred disassembly.

64-bit (sf == 1)

NGCS [<Xd>](#), [<Xm>](#)

is equivalent to

[SBCS](#) [<Xd>](#), XZR, [<Xm>](#)

and is always the preferred disassembly.

Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wm>

Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xm>

Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of [SBCS](#) gives the operational pseudocode for this instruction.

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

# NOP

No Operation does nothing, other than advance the value of the program counter by 4. This instruction can be used for instruction alignment purposes.

## Note

The timing effects of including a `NOP` instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, `NOP` instructions are not suitable for timing loops.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1
																CRm				op2												

## NOP

```
// Empty.
```

## Operation

```
return;    // do nothing
```

## Operational information

- If `PSTATE.DIT` is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the `NZCV` flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the `NZCV` flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORN (shifted register)

Bitwise OR NOT (shifted register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias [MVN](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
sf	0	1	0	1	0	1	0	shift	1	Rm						imm6						Rn						Rd					
opc								N																									

32-bit (sf == 0)

ORN <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

ORN <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR
- <amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.  
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Alias Conditions

Alias	Is preferred when
<a href="#">MVN</a>	Rn == '11111'

## Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bits(datasize) result;

operand2 = NOT(operand2);

result = operand1 OR operand2;
X[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



ORR (immediate)

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate register value, and writes the result to the destination register.

This instruction is used by the alias [MOV \(bitmask immediate\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	1	1	0	0	1	0	0	N	immr						imms						Rn						Rd			
opc																															

32-bit (sf == 0 && N == 0)

```
ORR <Wd|WSP>, <Wn>, #<imm>
```

64-bit (sf == 1)

```
ORR <Xd|SP>, <Xn>, #<imm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE, datasize);
```

Assembler Symbols

- <Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <imm> For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".  
For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Alias Conditions

Alias	Is preferred when
<a href="#">MOV (bitmask immediate)</a>	Rn == '11111' && ! <a href="#">MoveWidePreferred</a> (sf, N, imms, immr)

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];

result = operand1 OR imm;
if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORR (shifted register)

Bitwise OR (shifted register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias [MOV \(register\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
sf	0	1	0	1	0	1	0	shift	0	Rm						imm6						Rn						Rd									
opc										N																											

32-bit (sf == 0)

```
ORR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit (sf == 1)

```
ORR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>

Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm>

Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>

Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm>

Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <shift>

Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR
- <amount>

For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.  
  
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Alias Conditions

Alias	Is preferred when
<a href="#">MOV (register)</a>	shift == '00' && imm6 == '000000' && Rn == '11111'

## Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bits(datasize) result;

result = operand1 OR operand2;
X[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PACDA, PACDZA

Pointer Authentication Code for Data address, using key A. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key A.

The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDA.
- The value zero, for PACDZA.

Integer  
(FEAT\_PAuth)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	1	0	Rn					Rd				

PACDA (Z == 0)

```
PACDA <Xd>, <Xn|SP>
```

PACDZA (Z == 1 && Rn == 11111)

```
PACDZA <Xd>

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // PACDA
    if n == 31 then source_is_sp = TRUE;
else // PACDZA
    if n != 31 then UNDEFINED;
```

Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if source_is_sp then
    X[d, 64] = AddPACDA(X[d, 64], SP[]);
else
    X[d, 64] = AddPACDA(X[d, 64], X[n, 64]);
```

PACDB, PACDZB

Pointer Authentication Code for Data address, using key B. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key B.

The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDB.
- The value zero, for PACDZB.

Integer  
(FEAT\_PAuth)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	1	1	Rn				Rd					

PACDB (Z == 0)

```
PACDB <Xd>, <Xn|SP>
```

PACDZB (Z == 1 && Rn == 11111)

```
PACDZB <Xd>

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // PACDB
    if n == 31 then source_is_sp = TRUE;
else // PACDZB
    if n != 31 then UNDEFINED;
```

Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if source_is_sp then
    X[d, 64] = AddPACDB(X[d, 64], SP[]);
else
    X[d, 64] = AddPACDB(X[d, 64], X[n, 64]);
```

PACGA

Pointer Authentication Code, using Generic key. This instruction computes the pointer authentication code for a 64-bit value in the first source register, using a modifier in the second source register, and the Generic key. The computed pointer authentication code is written to the most significant 32 bits of the destination register, and the least significant 32 bits of the destination register are set to zero.

Integer  
(FEAT\_PAAuth)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	0	1	1	0	Rm					0	0	1	1	0	0	Rn					Rd				

PACGA <Xd>, <Xn>, <Xm|SP>

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if !HavePACExt() then
    UNDEFINED;

if m == 31 then source_is_sp = TRUE;
```

Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm|SP> Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Rm" field.

Operation

```
if source_is_sp then
    X[d, 64] = AddPACGA(X[n, 64], SP[]);
else
    X[d, 64] = AddPACGA(X[n, 64], X[m, 64]);
```

PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA

Pointer Authentication Code for Instruction address, using key A. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key A.

The address is:

- In the general-purpose register that is specified by <Xd> for PACIA and PACIZA.
- In X17, for PACIA1716.
- In X30, for PACIASP and PACIAZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACIA.
- The value zero, for PACIZA and PACIAZ.
- In X16, for PACIA1716.
- In SP, for PACIASP.

It has encodings from 2 classes: [Integer](#) and [System](#)

Integer  
(FEAT\_PAuth)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	0	0	Rn				Rd					

PACIA (Z == 0)

PACIA <Xd>, <Xn|SP>

PACIZA (Z == 1 && Rn == 11111)

PACIZA <Xd>

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACEExt() then
    UNDEFINED;

if Z == '0' then // PACIA
    if n == 31 then source_is_sp = TRUE;
else // PACIZA
    if n != 31 then UNDEFINED;
```

System  
(FEAT\_PAuth)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	x	1	0	0	x	1	1	1	1	1					
																					CRm				op2											



## PACIA1716 (CRm == 0001 && op2 == 000)

PACIA1716

## PACIASP (CRm == 0011 && op2 == 001)

PACIASP

## PACIAZ (CRm == 0011 && op2 == 000)

PACIAZ

```
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
  when '0011 000'    // PACIAZ
    d = 30;
    n = 31;
  when '0011 001'    // PACIASP
    d = 30;
    source_is_sp = TRUE;
    if HaveBTIExt\(\) then
      // Check for branch target compatibility between PSTATE.BTYPE
      // and implicit branch target of PACIASP instruction.
      SetBTypeCompatible\(BTypeCompatible\_PACIXSP\(\)\);

  when '0001 000'    // PACIA1716
    d = 17;
    n = 16;
  when '0001 010' SEE "PACIB";
  when '0001 100' SEE "AUTIA";
  when '0001 110' SEE "AUTIB";
  when '0011 01x' SEE "PACIB";
  when '0011 10x' SEE "AUTIA";
  when '0011 11x' SEE "AUTIB";
  when '0000 111' SEE "XPACLRI";
  otherwise SEE "HINT";
```

## Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

## Operation

```
if HavePACExt\(\) then
  if source_is_sp then
    X[d, 64] = AddPACIA(X[d, 64], SP[]);
  else
    X[d, 64] = AddPACIA(X[d, 64], X[n, 64]);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB

Pointer Authentication Code for Instruction address, using key B. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key B.

The address is:

- In the general-purpose register that is specified by <Xd> for PACIB and PACIZB.
- In X17, for PACIB1716.
- In X30, for PACIBSP and PACIBZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACIB.
- The value zero, for PACIZB and PACIBZ.
- In X16, for PACIB1716.
- In SP, for PACIBSP.

It has encodings from 2 classes: [Integer](#) and [System](#)

Integer  
(FEAT\_PAuth)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	Z	0	0	1	Rn				Rd					

PACIB (Z == 0)

PACIB <Xd>, <Xn|SP>

PACIZB (Z == 1 && Rn == 11111)

PACIZB <Xd>

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACEExt() then
    UNDEFINED;

if Z == '0' then // PACIB
    if n == 31 then source_is_sp = TRUE;
else // PACIZB
    if n != 31 then UNDEFINED;
```

System  
(FEAT\_PAuth)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	x	1	0	1	x	1	1	1	1	1				
																				CRm				op2											

## PACIB1716 (CRm == 0001 && op2 == 010)

PACIB1716

## PACIBSP (CRm == 0011 && op2 == 011)

PACIBSP

## PACIBZ (CRm == 0011 && op2 == 010)

PACIBZ

```
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
  when '0011 010'    // PACIBZ
    d = 30;
    n = 31;
  when '0011 011'    // PACIBSP
    d = 30;
    source_is_sp = TRUE;
    if HaveBTIExt() then
      // Check for branch target compatibility between PSTATE.BTYPE
      // and implicit branch target of PACIBSP instruction.
      SetBTypeCompatible(BTypeCompatible\_PACIXSP());
  when '0001 010'    // PACIB1716
    d = 17;
    n = 16;
  when '0001 000' SEE "PACIA";
  when '0001 100' SEE "AUTIA";
  when '0001 110' SEE "AUTIB";
  when '0011 00x' SEE "PACIA";
  when '0011 10x' SEE "AUTIA";
  when '0011 11x' SEE "AUTIB";
  when '0000 111' SEE "XPACLRI";
  otherwise SEE "HINT";
```

## Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

## Operation

```
if HavePACExt() then
  if source_is_sp then
    X[d, 64] = AddPACIB(X[d, 64], SP[]);
  else
    X[d, 64] = AddPACIB(X[d, 64], X[n, 64]);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

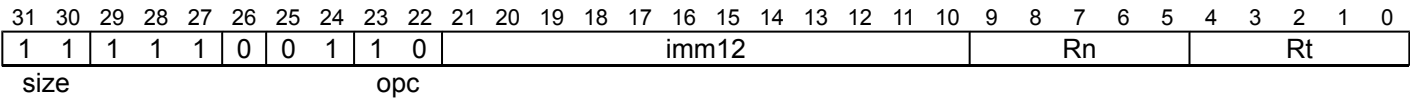
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PRFM (immediate)

Prefetch Memory (immediate) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of a P<sub>RFM</sub> instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/Store addressing modes*.



```
PRFM (<prfop>|#<imm5>), [<Xn|SP>{, #<pimm>}]
```

```
bits(64) offset = LSL(ZeroExtend(imm12, 64), 3);
```

Assembler Symbols

- <prfop> Is the prefetch operation, defined as <type><target><policy>.  
<type> is one of:  
**PLD** Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.  
**PLI** Preload instructions, encoded in the "Rt<4:3>" field as 0b01.  
**PST** Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.  
<target> is one of:  
**L1** Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.  
**L2** Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.  
**L3** Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.  
**SLC** When FEAT\_PRFM\_SLC is implemented, system level cache, encoded in the "Rt<2:1>" field as 0b11.  
<policy> is one of:  
**KEEP** Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.  
**STRM** Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.  
  
For more information on these prefetch operations, see *Prefetch memory*.  
For other encodings of the "Rt" field, use <imm5>.
- <imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <pimm> Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

## Shared Decode

```
integer n = UInt(Rn);  
integer t = UInt(Rt);
```

## Operation

```
bits(64) address;  
  
if n == 31 then  
    address = SP[];  
else  
    address = X[n, 64];  
  
address = address + offset;  
  
Prefetch(address, t<4:0>);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

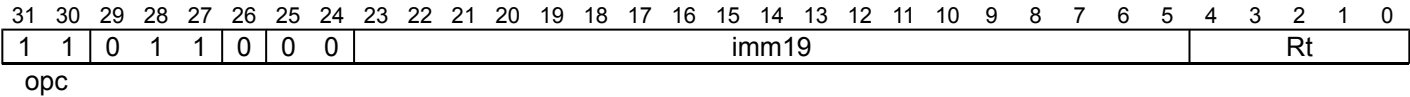
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PRFM (literal)

Prefetch Memory (literal) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of a `PRFM` instruction is IMPLEMENTATION DEFINED. For more information, see [Prefetch memory](#).

For information about memory accesses, see [Load/Store addressing modes](#).



```
PRFM (<prfop>|#<imm5>), <label>
```

```
integer t = UInt(Rt);
bits(64) offset;

offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

- <prfop> Is the prefetch operation, defined as <type><target><policy>. <type> is one of:  
**PLD** Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.  
**PLI** Preload instructions, encoded in the "Rt<4:3>" field as 0b01.  
**PST** Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.
- <target> is one of:  
**L1** Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.  
**L2** Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.  
**L3** Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.  
**SLC** When FEAT\_PRFM\_SLC is implemented, system level cache, encoded in the "Rt<2:1>" field as 0b11.
- <policy> is one of:  
**KEEP** Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.  
**STRM** Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.
- For more information on these prefetch operations, see [Prefetch memory](#).  
For other encodings of the "Rt" field, use <imm5>.
- <imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.
- <label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

## Operation

```
bits(64) address = PC[] + offset;  
Prefetch(address, t<4:0>);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PRFM (register)

Prefetch Memory (register) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of a P<sub>RFM</sub> instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	1	Rm				x	1	x	S	1	0	Rn				!= 11xxx						
size				opc							option										Rt										

P<sub>RFM</sub> (<prfop>|<#imm5>), [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

```
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 3 else 0;
```

Assembler Symbols

- <prfop>

Is the prefetch operation, defined as <type><target><policy>. <type> is one of:  
**PLD**  
Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.  
**PLI**  
Preload instructions, encoded in the "Rt<4:3>" field as 0b01.  
**PST**  
Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.  
<target> is one of:  
**L1**  
Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.  
**L2**  
Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.  
**L3**  
Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.  
**SLC**  
When FEAT\_PRFM\_SLC is implemented, system level cache, encoded in the "Rt<2:1>" field as 0b11.  
<policy> is one of:  
**KEEP**  
Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.  
**STRM**  
Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.  
  
For more information on these prefetch operations, see *Prefetch memory*.  
For other encodings of the "Rt" field, use <imm5>.
- <imm5>

Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm>

When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm>

When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.



<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

if n == 31 then
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

Prefetch(address, t<4:0>);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

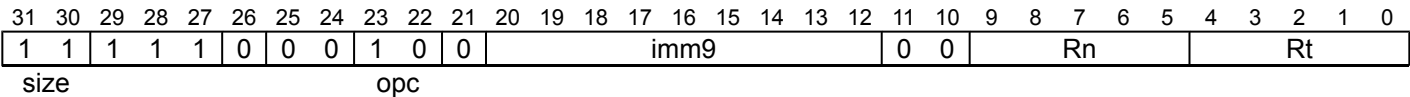
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PRFUM

Prefetch Memory (unscaled offset) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of a PRFUM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

For information about memory accesses, see Load/Store addressing modes.



PRFUM (<prfop>|<imm5>), [<Xn|SP>{, #<sim>}]

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

- <prfop>

Is the prefetch operation, defined as <type><target><policy>.  
<type> is one of:  
**PLD**  
Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.  
**PLI**  
Preload instructions, encoded in the "Rt<4:3>" field as 0b01.  
**PST**  
Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.  
<target> is one of:  
**L1**  
Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.  
**L2**  
Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.  
**L3**  
Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.  
<policy> is one of:  
**KEEP**  
Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.  
**STRM**  
Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.  
  
For more information on these prefetch operations, see Prefetch memory.  
For other encodings of the "Rt" field, use <imm5>.
- <imm5>

Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.  
This syntax is only for encodings that are not accessible using <prfop>.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim>

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);  
integer t = UInt(Rt);
```

## Operation

```
bits(64) address;  
  
if n == 31 then  
    address = SP[];  
else  
    address = X[n, 64];  
  
address = address + offset;  
  
Prefetch(address, t<4:0>);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PSB CSYNC

Profiling Synchronization Barrier. This instruction is a barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following DSB instruction completes when the writes to the profiling buffer have completed.

If the Statistical Profiling Extension is not implemented, this instruction executes as a NOP.

System  
(FEAT\_SPE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	1	0	0	0	1	1	1	1	1	1
																CRm				op2											

PSB CSYNC

```
if !HaveStatisticalProfiling() then EndOfInstruction();
```

Operation

```
ProfilingSynchronizationBarrier();
```

PSSBB

Physical Speculative Store Bypass Barrier is a memory barrier that prevents speculative loads from bypassing earlier stores to the same physical address under certain conditions. For more information and details of the semantics, see *Physical Speculative Store Bypass Barrier (PSSBB)*.

This is an alias of [DSB](#). This means:

- The encodings in this description are named to match the encodings of [DSB](#).
- The description of [DSB](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	1	1	1
																CRm				opc											

PSSBB

is equivalent to

[DSB](#) #4

and is always the preferred disassembly.

Operation

The description of [DSB](#) gives the operational pseudocode for this instruction.

# RBIT

Reverse Bits reverses the bit order in a register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	Rn				Rd					

## 32-bit (sf == 0)

RBIT <Wd>, <Wn>

## 64-bit (sf == 1)

RBIT <Xd>, <Xn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;
```

## Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

## Operation

```
bits(datasize) operand = X[n, datasize];
bits(datasize) result;

for i = 0 to datasize-1
    result<(datasize-1)-i> = operand<i>;

X[d, datasize] = result;
```

## Operational information

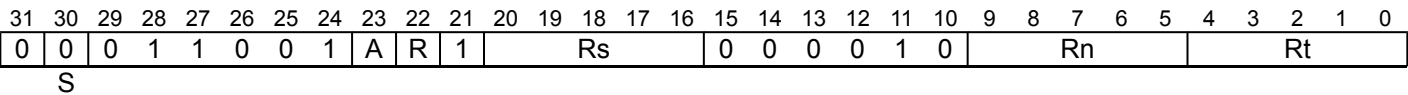
- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

# RCWCAS, RCWCASA, RCWCASL, RCWCASAL

Read Check Write Compare and Swap doubleword in memory reads a 64-bit doubleword from memory, and compares it against the value held in a register. If the comparison is equal, the value in a second register is conditionally written to memory. Storing back to memory is conditional on RCW Checks. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the update of memory.

- RCWCASA and RCWCASAL load from memory with acquire semantics.
- RCWCASL and RCWCASAL store to memory with release semantics.
- RCWCAS has neither acquire nor release semantics.

## Integer (FEAT\_THE)



### RCWCAS (A == 0 && R == 0)

RCWCAS <Xs>, <Xt>, [<Xn|SP>]

### RCWCASA (A == 1 && R == 0)

RCWCASA <Xs>, <Xt>, [<Xn|SP>]

### RCWCASAL (A == 1 && R == 1)

RCWCASAL <Xs>, <Xt>, [<Xn|SP>]

### RCWCASL (A == 0 && R == 1)

RCWCASL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveTHExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

## Assembler Symbols

- <Xs> Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
- <Xt> Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) newdata = X[t, 64];
bits(64) compdata = X[s, 64];
bits(64) readdata;
bits(4) nzcvc;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp\_CAS, FALSE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

(nzcvc, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcvc;
X[s, 64] = readdata;    // Return the old value when s!=31
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## RCWCASP, RCWCASPA, RCWCASPL, RCWCASPAL

Read Check Write Compare and Swap quadword in memory reads a 128-bit quadword from memory, and compares it against the value held in a pair of registers. If the comparison is equal, the value in a second pair of registers is conditionally written to memory. Storing back to memory is conditional on RCW Checks. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the update of memory.

- RCWCASPA and RCWCASPAL load from memory with acquire semantics.
- RCWCASPL and RCWCASPAL store to memory with release semantics.
- RCWCASP has neither acquire nor release semantics.

### Integer

(FEAT\_D128 && FEAT\_THE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	1	A	R	1	Rs				0	0	0	0	1	1	Rn				Rt						
S																															

### RCWCASP (A == 0 && R == 0)

RCWCASP <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]

### RCWCASPA (A == 1 && R == 0)

RCWCASPA <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]

### RCWCASPAL (A == 1 && R == 1)

RCWCASPAL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]

### RCWCASPL (A == 0 && R == 1)

RCWCASPL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]

```
if !Have128BitDescriptorExt() || !HaveTHExt() then UNDEFINED;
if Rs<0> == '1' then UNDEFINED;
if Rt<0> == '1' then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

### Assembler Symbols

- <Xs> Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Xs> must be an even-numbered register.
- <X(s+1)> Is the 64-bit name of the second general-purpose register to be compared and loaded.
- <Xt> Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Xt> must be an even-numbered register.
- <X(t+1)> Is the 64-bit name of the second general-purpose register to be conditionally stored.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(128) newdata;
bits(128) compdata;
bits(128) readdata;
bits(4) nzcvc;

bits(64) s1 = X[s, 64];
bits(64) s2 = X[s+1, 64];
bits(64) t1 = X[t, 64];
bits(64) t2 = X[t+1, 64];

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_CAS, FALSE, acquire, release, tagchecked);

compdata = if BigEndian(accdesc.acctype) then s1:s2 else s2:s1;
newdata = if BigEndian(accdesc.acctype) then t1:t2 else t2:t1;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

(nzcvc, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcvc;
if BigEndian(accdesc.acctype) then
    X[s, 64] = readdata<127:64>;
    X[s+1, 64] = readdata<63:0>;
else
    X[s, 64] = readdata<63:0>;
    X[s+1, 64] = readdata<127:64>;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

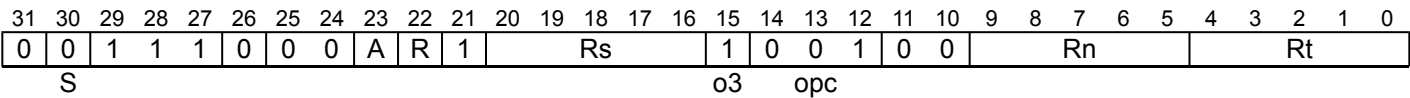
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# RCWCLR, RCWCLRA, RCWCLRL, RCWCLRAL

Read Check Write atomic bit Clear on doubleword in memory atomically loads a 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the destination register. This instruction updates the condition flags based on the result of the update of memory.

- RCWCLRA and RCWCLRAL load from memory with acquire semantics.
- RCWCLRL and RCWCLRAL store to memory with release semantics.
- RCWCLR has neither acquire nor release semantics.

## Integer (FEAT\_THE)



### RCWCLR (A == 0 && R == 0)

```
RCWCLR <Xs>, <Xt>, [<Xn|SP>]
```

### RCWCLRA (A == 1 && R == 0)

```
RCWCLRA <Xs>, <Xt>, [<Xn|SP>]
```

### RCWCLRAL (A == 1 && R == 1)

```
RCWCLRAL <Xs>, <Xt>, [<Xn|SP>]
```

### RCWCLRL (A == 0 && R == 1)

```
RCWCLRL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveTHExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

## Assembler Symbols

- <Xs> Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
- <Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) newdata = X[s, 64];
bits(64) readdata;
bits(4) nzcv;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp\_BIC, FALSE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(64) compdata = bits(64) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
X[t, 64] = readdata;    // Return the old value when t!=31
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

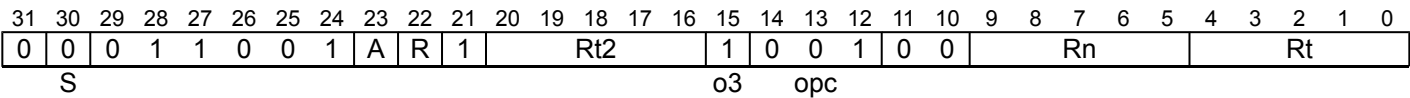
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# RCWCLRP, RCWCLRPA, RCWCLRPL, RCWCLRPA

Read Check Write atomic bit Clear on quadword in memory atomically loads a 128-bit quadword from memory, performs a bitwise AND with the complement of the value held in a pair of registers on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the same pair of registers. This instruction updates the condition flags based on the result of the update of memory.

- RCWCLRPA and RCWCLRPA load from memory with acquire semantics.
- RCWCLRPL and RCWCLRPA store to memory with release semantics.
- RCWCLRP has neither acquire nor release semantics.

## Integer (FEAT\_D128 && FEAT\_THE)



### RCWCLRP (A == 0 && R == 0)

```
RCWCLRP <Xt1>, <Xt2>, [<Xn|SP>]
```

### RCWCLRPA (A == 1 && R == 0)

```
RCWCLRPA <Xt1>, <Xt2>, [<Xn|SP>]
```

### RCWCLRPA (A == 1 && R == 1)

```
RCWCLRPA <Xt1>, <Xt2>, [<Xn|SP>]
```

### RCWCLRPL (A == 0 && R == 1)

```
RCWCLRPL <Xt1>, <Xt2>, [<Xn|SP>]
```

```
if !Have128BitDescriptorExt() || !HaveTHExt() then UNDEFINED;
if Rt == '11111' then UNDEFINED;
if Rt2 == '11111' then UNDEFINED;
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer n = UInt(Rn);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Assembler Symbols

- <Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) value1;
bits(64) value2;
bits(128) newdata;
bits(128) readdata;
bits(4) nzcvc;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp\_BIC, FALSE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

value1 = X[t, 64];
value2 = X[t2, 64];

newdata = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

bits(128) compdata = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
(nzcvc, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcvc;
if rt_unknown then
    readdata = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = readdata<127:64>;
    X[t2, 64] = readdata<63:0>;
else
    X[t, 64] = readdata<63:0>;
    X[t2, 64] = readdata<127:64>;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# RCWSCAS, RCWSCASA, RCWSCASL, RCWSCASAL

Read Check Write Software Compare and Swap doubleword in memory reads a 64-bit doubleword from memory, and compares it against the value held in a register. If the comparison is equal, the value in a second register is conditionally written to memory. Storing back to memory is conditional on RCW Checks and RCWS Checks. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the update of memory.

- RCWSCASA and RCWSCASAL load from memory with acquire semantics.
- RCWSCASL and RCWSCASAL store to memory with release semantics.
- RCWSCAS has neither acquire nor release semantics.

## Integer (FEAT\_THE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	1	A	R	1	Rs					0	0	0	0	1	0	Rn					Rt				
S																															

### RCWSCAS (A == 0 && R == 0)

RCWSCAS <Xs>, <Xt>, [<Xn|SP>]

### RCWSCASA (A == 1 && R == 0)

RCWSCASA <Xs>, <Xt>, [<Xn|SP>]

### RCWSCASAL (A == 1 && R == 1)

RCWSCASAL <Xs>, <Xt>, [<Xn|SP>]

### RCWSCASL (A == 0 && R == 1)

RCWSCASL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveTHExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

## Assembler Symbols

- <Xs> Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
- <Xt> Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) newdata = X[t, 64];
bits(64) compdata = X[s, 64];
bits(64) readdata;
bits(4) nzcvc;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp\_CAS, TRUE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

(nzcvc, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcvc;
X[s, 64] = readdata;    // Return the old value when s!=31
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# RCWSCASP, RCWSCASPA, RCWSCASPL, RCWSCASPAL

Read Check Write Software Compare and Swap quadword in memory reads a 128-bit quadword from memory, and compares it against the value held in a pair of registers. If the comparison is equal, the value in a second pair of registers is conditionally written to memory. Storing back to memory is conditional on RCW Checks and RCWS Checks. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the update of memory.

- RCWSCASPA and RCWSCASPAL load from memory with acquire semantics.
- RCWSCASPL and RCWSCASPAL store to memory with release semantics.
- RCWSCASP has neither acquire nor release semantics.

## Integer (FEAT\_D128 && FEAT\_THE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	1	A	R	1	Rs					0	0	0	0	1	1	Rn					Rt				
S																															

### RCWSCASP (A == 0 && R == 0)

RCWSCASP <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]

### RCWSCASPA (A == 1 && R == 0)

RCWSCASPA <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]

### RCWSCASPAL (A == 1 && R == 1)

RCWSCASPAL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]

### RCWSCASPL (A == 0 && R == 1)

RCWSCASPL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]

```
if !Have128BitDescriptorExt() || !HaveTHExt() then UNDEFINED;
if Rs<0> == '1' then UNDEFINED;
if Rt<0> == '1' then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

## Assembler Symbols

- <Xs> Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Xs> must be an even-numbered register.
- <X(s+1)> Is the 64-bit name of the second general-purpose register to be compared and loaded.
- <Xt> Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Xt> must be an even-numbered register.
- <X(t+1)> Is the 64-bit name of the second general-purpose register to be conditionally stored.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(128) newdata;
bits(128) compdata;
bits(128) readdata;
bits(4) nzcvc;

bits(64) s1 = X[s, 64];
bits(64) s2 = X[s+1, 64];
bits(64) t1 = X[t, 64];
bits(64) t2 = X[t+1, 64];

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_CAS, TRUE, acquire, release, tagchecked);

compdata = if BigEndian(accdesc.acctype) then s1:s2 else s2:s1;
newdata = if BigEndian(accdesc.acctype) then t1:t2 else t2:t1;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

(nzcvc, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcvc;
if BigEndian(accdesc.acctype) then
    X[s, 64] = readdata<127:64>;
    X[s+1, 64] = readdata<63:0>;
else
    X[s, 64] = readdata<63:0>;
    X[s+1, 64] = readdata<127:64>;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

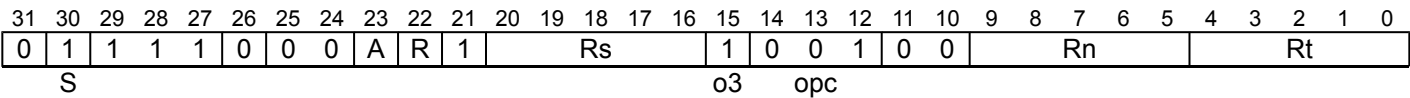
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## RCWSCLR, RCWSCLRA, RCWSCLRL, RCWSCLRAL

Read Check Write Software atomic bit Clear on doubleword in memory atomically loads a 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the destination register. This instruction updates the condition flags based on the result of the update of memory.

- RCWSCLRA and RCWSCLRAL load from memory with acquire semantics.
- RCWSCLRL and RCWSCLRAL store to memory with release semantics.
- RCWSCLR has neither acquire nor release semantics.

### Integer (FEAT\_THE)



#### RCWSCLR (A == 0 && R == 0)

RCWSCLR <Xs>, <Xt>, [<Xn|SP>]

#### RCWSCLRA (A == 1 && R == 0)

RCWSCLRA <Xs>, <Xt>, [<Xn|SP>]

#### RCWSCLRAL (A == 1 && R == 1)

RCWSCLRAL <Xs>, <Xt>, [<Xn|SP>]

#### RCWSCLRL (A == 0 && R == 1)

RCWSCLRL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveTHExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

### Assembler Symbols

- <Xs>            Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
- <Xt>            Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP>        Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) newdata = X[s, 64];
bits(64) readdata;
bits(4) nzcvc;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp\_BIC, TRUE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(64) compdata = bits(64) UNKNOWN;    // Irrelevant when not executing CAS
(nzcvc, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcvc;
X[t, 64] = readdata;    // Return the old value when t!=31
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

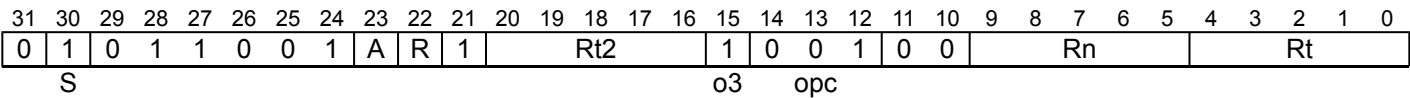
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# RCWSCLRP, RCWSCLRPA, RCWSCLRPL, RCWSCLRPAL

Read Check Write Software atomic bit Clear on quadword in memory atomically loads a 128-bit quadword from memory, performs a bitwise AND with the complement of the value held in a pair of registers on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the same pair of registers. This instruction updates the condition flags based on the result of the update of memory.

- RCWSCLRPA and RCWSCLRPAL load from memory with acquire semantics.
- RCWSCLRPL and RCWSCLRPAL store to memory with release semantics.
- RCWSCLRP has neither acquire nor release semantics.

## Integer (FEAT\_D128 && FEAT\_THE)



### RCWSCLRP (A == 0 && R == 0)

```
RCWSCLRP <Xt1>, <Xt2>, [<Xn|SP>]
```

### RCWSCLRPA (A == 1 && R == 0)

```
RCWSCLRPA <Xt1>, <Xt2>, [<Xn|SP>]
```

### RCWSCLRPAL (A == 1 && R == 1)

```
RCWSCLRPAL <Xt1>, <Xt2>, [<Xn|SP>]
```

### RCWSCLRPL (A == 0 && R == 1)

```
RCWSCLRPL <Xt1>, <Xt2>, [<Xn|SP>]
```

```
if !Have128BitDescriptorExt() || !HaveTHExt() then UNDEFINED;
if Rt == '11111' then UNDEFINED;
if Rt2 == '11111' then UNDEFINED;
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer n = UInt(Rn);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Assembler Symbols

- <Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) value1;
bits(64) value2;
bits(128) newdata;
bits(128) readdata;
bits(4) nzcvc;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp\_BIC, TRUE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

value1 = X[t, 64];
value2 = X[t2, 64];

newdata = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

bits(128) compdata = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
(nzcvc, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcvc;
if rt_unknown then
    readdata = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = readdata<127:64>;
    X[t2, 64] = readdata<63:0>;
else
    X[t, 64] = readdata<63:0>;
    X[t2, 64] = readdata<127:64>;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

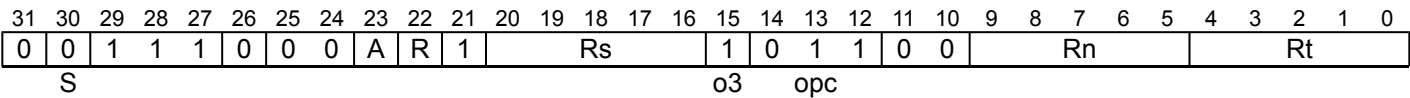
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# RCWSET, RCWSETA, RCWSETL, RCWSETAL

Read Check Write atomic bit Set on doubleword in memory atomically loads a 64-bit doubleword from memory, performs a bitwise OR with the complement of the value held in a register on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the destination register. This instruction updates the condition flags based on the result of the update of memory.

- RCWSETA and RCWSETAL load from memory with acquire semantics.
- RCWSETL and RCWSETAL store to memory with release semantics.
- RCWSET has neither acquire nor release semantics.

## Integer (FEAT\_THE)



### RCWSET (A == 0 && R == 0)

```
RCWSET <Xs>, <Xt>, [<Xn|SP>]
```

### RCWSETA (A == 1 && R == 0)

```
RCWSETA <Xs>, <Xt>, [<Xn|SP>]
```

### RCWSETAL (A == 1 && R == 1)

```
RCWSETAL <Xs>, <Xt>, [<Xn|SP>]
```

### RCWSETL (A == 0 && R == 1)

```
RCWSETL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveTHExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

## Assembler Symbols

- <Xs> Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
- <Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) newdata = X[s, 64];
bits(64) readdata;
bits(4) nzcvc;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp\_ORR, FALSE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(64) compdata = bits(64) UNKNOWN;    // Irrelevant when not executing CAS
(nzcvc, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcvc;
X[t, 64] = readdata;    // Return the old value when t!=31
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# RCWSETP, RCWSETPA, RCWSETPL, RCWSETPAL

Read Check Write atomic bit Set on quadword in memory atomically loads a 128-bit quadword from memory, performs a bitwise OR with the value held in a pair of registers on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the same pair of registers. This instruction updates the condition flags based on the result of the update of memory.

- RCWSETPA and RCWSETPAL load from memory with acquire semantics.
- RCWSETPL and RCWSETPAL store to memory with release semantics.
- RCWSETP has neither acquire nor release semantics.

## Integer (FEAT\_D128 && FEAT\_THE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	1	A	R	1	Rt2					1	0	1	1	0	0	Rn					Rt				
S											o3											opc									

### RCWSETP (A == 0 && R == 0)

RCWSETP <Xt1>, <Xt2>, [<Xn|SP>]

### RCWSETPA (A == 1 && R == 0)

RCWSETPA <Xt1>, <Xt2>, [<Xn|SP>]

### RCWSETPAL (A == 1 && R == 1)

RCWSETPAL <Xt1>, <Xt2>, [<Xn|SP>]

### RCWSETPL (A == 0 && R == 1)

RCWSETPL <Xt1>, <Xt2>, [<Xn|SP>]

```
if !Have128BitDescriptorExt() || !HaveTHExt() then UNDEFINED;
if Rt == '11111' then UNDEFINED;
if Rt2 == '11111' then UNDEFINED;
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer n = UInt(Rn);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable LSE128OVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

## Assembler Symbols

- <Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) value1;
bits(64) value2;
bits(128) newdata;
bits(128) readdata;
bits(4) nzcvc;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_ORR, FALSE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

value1 = X[t, 64];
value2 = X[t2, 64];

newdata = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

bits(128) compdata = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
(nzcvc, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcvc;
if rt_unknown then
    readdata = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = readdata<127:64>;
    X[t2, 64] = readdata<63:0>;
else
    X[t, 64] = readdata<63:0>;
    X[t2, 64] = readdata<127:64>;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

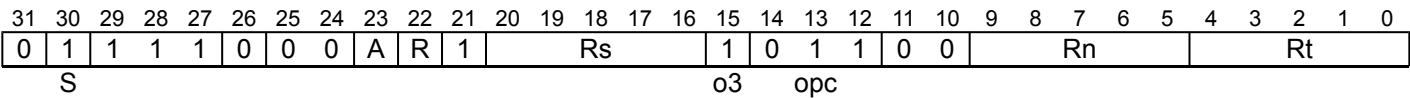
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## RCWSSET, RCWSSETA, RCWSSETL, RCWSSETAL

Read Check Write Software atomic bit Set on doubleword in memory atomically loads a 64-bit doubleword from memory, performs a bitwise OR with the complement of the value held in a register on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the destination register. This instruction updates the condition flags based on the result of the update of memory.

- RCWSSETA and RCWSSETAL load from memory with acquire semantics.
- RCWSSETL and RCWSSETAL store to memory with release semantics.
- RCWSSET has neither acquire nor release semantics.

### Integer (FEAT\_THE)



#### RCWSSET (A == 0 && R == 0)

RCWSSET <Xs>, <Xt>, [<Xn|SP>]

#### RCWSSETA (A == 1 && R == 0)

RCWSSETA <Xs>, <Xt>, [<Xn|SP>]

#### RCWSSETAL (A == 1 && R == 1)

RCWSSETAL <Xs>, <Xt>, [<Xn|SP>]

#### RCWSSETL (A == 0 && R == 1)

RCWSSETL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveTHExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

### Assembler Symbols

- <Xs> Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
- <Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) newdata = X[s, 64];
bits(64) readdata;
bits(4) nzcv;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp\_ORR, TRUE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(64) compdata = bits(64) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
X[t, 64] = readdata;    // Return the old value when t!=31
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

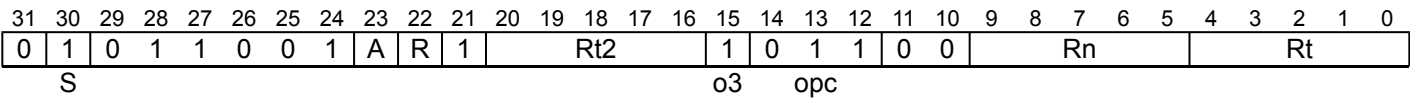
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# RCWSSETP, RCWSSETPA, RCWSSETPL, RCWSSETPAL

Read Check Write Software atomic bit Set on quadword in memory atomically loads a 128-bit quadword from memory, performs a bitwise OR with the value held in a pair of registers on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the same pair of registers. This instruction updates the condition flags based on the result of the update of memory.

- RCWSSETPA and RCWSSETPAL load from memory with acquire semantics.
- RCWSSETPL and RCWSSETPAL store to memory with release semantics.
- RCWSSETP has neither acquire nor release semantics.

## Integer (FEAT\_D128 && FEAT\_THE)



### RCWSSETP (A == 0 && R == 0)

```
RCWSSETP <Xt1>, <Xt2>, [<Xn|SP>]
```

### RCWSSETPA (A == 1 && R == 0)

```
RCWSSETPA <Xt1>, <Xt2>, [<Xn|SP>]
```

### RCWSSETPAL (A == 1 && R == 1)

```
RCWSSETPAL <Xt1>, <Xt2>, [<Xn|SP>]
```

### RCWSSETPL (A == 0 && R == 1)

```
RCWSSETPL <Xt1>, <Xt2>, [<Xn|SP>]
```

```
if !Have128BitDescriptorExt() || !HaveTHExt() then UNDEFINED;
if Rt == '11111' then UNDEFINED;
if Rt2 == '11111' then UNDEFINED;
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer n = UInt(Rn);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

## Assembler Symbols

- <Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) value1;
bits(64) value2;
bits(128) newdata;
bits(128) readdata;
bits(4) nzcvc;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_ORR, TRUE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

value1 = X[t, 64];
value2 = X[t2, 64];

newdata = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

bits(128) compdata = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
(nzcvc, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcvc;
if rt_unknown then
    readdata = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = readdata<127:64>;
    X[t2, 64] = readdata<63:0>;
else
    X[t, 64] = readdata<63:0>;
    X[t2, 64] = readdata<127:64>;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## RCWSSWP, RCWSSWPA, RCWSSWPL, RCWSSWPAL

Read Check Write Software Swap doubleword in memory atomically loads a 64-bit doubleword from a memory location, and conditionally stores the value held in a register back to the same memory location. Storing back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the destination register. This instruction updates the condition flags based on the result of the update of memory.

- RCWSSWPA and RCWSSWPAL load from memory with acquire semantics.
- RCWSSWPL and RCWSSWPAL store to memory with release semantics.
- RCWSSWP has neither acquire nor release semantics.

### Integer (FEAT\_THE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs				1	0	1	0	0	0	Rn				Rt						
S										o3				opc																	

#### RCWSSWP (A == 0 && R == 0)

RCWSSWP <Xs>, <Xt>, [<Xn|SP>]

#### RCWSSWPA (A == 1 && R == 0)

RCWSSWPA <Xs>, <Xt>, [<Xn|SP>]

#### RCWSSWPAL (A == 1 && R == 1)

RCWSSWPAL <Xs>, <Xt>, [<Xn|SP>]

#### RCWSSWPL (A == 0 && R == 1)

RCWSSWPL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveTHExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

### Assembler Symbols

- <Xs> Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
- <Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) newdata = X[s, 64];
bits(64) readdata;
bits(4) nzcvc;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp\_SWP, TRUE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(64) compdata = bits(64) UNKNOWN;    // Irrelevant when not executing CAS
(nzcvc, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcvc;
X[t, 64] = readdata;    // Return the old value when t!=31
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# RCWSSWPP, RCWSSWPPA, RCWSSWPPL, RCWSSWPPAL

Read Check Write Software Swap quadword in memory atomically loads a 128-bit quadword from a memory location, and conditionally stores the value held in a pair of registers back to the same memory location. Storing back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the same pair of registers. This instruction updates the condition flags based on the result of the update of memory.

- RCWSSWPPA and RCWSSWPPAL load from memory with acquire semantics.
- RCWSSWPPL and RCWSSWPPAL store to memory with release semantics.
- RCWSSWPP has neither acquire nor release semantics.

## Integer (FEAT\_D128 && FEAT\_THE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	1	A	R	1	Rt2					1	0	1	0	0	0	Rn					Rt				
S											o3										opc										

### RCWSSWPP (A == 0 && R == 0)

RCWSSWPP <Xt1>, <Xt2>, [<Xn|SP>]

### RCWSSWPPA (A == 1 && R == 0)

RCWSSWPPA <Xt1>, <Xt2>, [<Xn|SP>]

### RCWSSWPPAL (A == 1 && R == 1)

RCWSSWPPAL <Xt1>, <Xt2>, [<Xn|SP>]

### RCWSSWPPL (A == 0 && R == 1)

RCWSSWPPL <Xt1>, <Xt2>, [<Xn|SP>]

```
if !Have128BitDescriptorExt() || !HaveTHExt() then UNDEFINED;
if Rt == '11111' then UNDEFINED;
if Rt2 == '11111' then UNDEFINED;
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer n = UInt(Rn);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable LSE128OVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

## Assembler Symbols

- <Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) value1;
bits(64) value2;
bits(128) newdata;
bits(128) readdata;
bits(4) nzcvc;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp\_SWP, TRUE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

value1 = X[t, 64];
value2 = X[t2, 64];

newdata = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

bits(128) compdata = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
(nzcvc, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcvc;
if rt_unknown then
    readdata = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = readdata<127:64>;
    X[t2, 64] = readdata<63:0>;
else
    X[t, 64] = readdata<63:0>;
    X[t2, 64] = readdata<127:64>;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

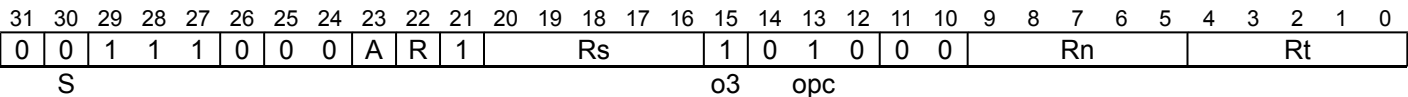
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## RCSWP, RCSWPA, RCWSWPL, RCWSWPAL

Read Check Write Swap doubleword in memory atomically loads a 64-bit doubleword from a memory location, and conditionally stores the value held in a register back to the same memory location. Storing back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the destination register. This instruction updates the condition flags based on the result of the update of memory.

- RCWSWPA and RCWSWPAL load from memory with acquire semantics.
- RCWSWPL and RCWSWPAL store to memory with release semantics.
- RCSWP has neither acquire nor release semantics.

### Integer (FEAT\_THE)



#### RCSWP (A == 0 && R == 0)

RCSWP <Xs>, <Xt>, [<Xn|SP>]

#### RCSWPA (A == 1 && R == 0)

RCSWPA <Xs>, <Xt>, [<Xn|SP>]

#### RCWSWPAL (A == 1 && R == 1)

RCWSWPAL <Xs>, <Xt>, [<Xn|SP>]

#### RCWSWPL (A == 0 && R == 1)

RCWSWPL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveTHExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

### Assembler Symbols

- <Xs> Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
- <Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) newdata = X[s, 64];
bits(64) readdata;
bits(4) nzcvc;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp\_SWP, FALSE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(64) compdata = bits(64) UNKNOWN;    // Irrelevant when not executing CAS
(nzcvc, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcvc;
X[t, 64] = readdata;    // Return the old value when t!=31
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# RCWSWPP, RCWSWPPA, RCWSWPPL, RCWSWPPAL

Read Check Write Swap quadword in memory atomically loads a 128-bit quadword from a memory location, and conditionally stores the value held in a pair of registers back to the same memory location. Storing back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the same pair of registers. This instruction updates the condition flags based on the result of the update of memory.

- RCWSWPPA and RCWSWPPAL load from memory with acquire semantics.
- RCWSWPPL and RCWSWPPAL store to memory with release semantics.
- RCWSWPP has neither acquire nor release semantics.

## Integer (FEAT\_D128 && FEAT\_THE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	1	A	R	1	Rt2					1	0	1	0	0	0	Rn					Rt				
S										o3										opc											

### RCWSWPP (A == 0 && R == 0)

```
RCWSWPP <Xt1>, <Xt2>, [<Xn|SP>]
```

### RCWSWPPA (A == 1 && R == 0)

```
RCWSWPPA <Xt1>, <Xt2>, [<Xn|SP>]
```

### RCWSWPPAL (A == 1 && R == 1)

```
RCWSWPPAL <Xt1>, <Xt2>, [<Xn|SP>]
```

### RCWSWPPL (A == 0 && R == 1)

```
RCWSWPPL <Xt1>, <Xt2>, [<Xn|SP>]
```

```
if !Have128BitDescriptorExt() || !HaveTHExt() then UNDEFINED;
if Rt == '11111' then UNDEFINED;
if Rt2 == '11111' then UNDEFINED;
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer n = UInt(Rn);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable LSE128OVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

## Assembler Symbols

- <Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) value1;
bits(64) value2;
bits(128) newdata;
bits(128) readdata;
bits(4) nzcvc;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_SWP, FALSE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

value1 = X[t, 64];
value2 = X[t2, 64];

newdata = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

bits(128) compdata = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
(nzcvc, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcvc;
if rt_unknown then
    readdata = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = readdata<127:64>;
    X[t2, 64] = readdata<63:0>;
else
    X[t, 64] = readdata<63:0>;
    X[t2, 64] = readdata<127:64>;
```

## Operational information

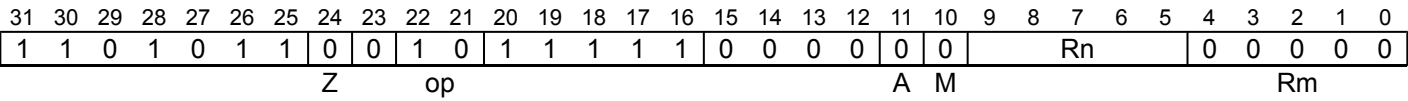
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RET

Return from subroutine branches unconditionally to an address in a register, with a hint that this is a subroutine return.



RET {<Xn>}

```
integer n = UInt(Rn);
```

Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field. Defaults to X30 if absent.

Operation

```
bits(64) target = X[n, 64];

if HaveGCS() && GCSPCREnabled(PSTATE.EL) && n != 16 then
    target = LoadCheckGCSRecord(target, GCSInstType_PRET);
    SetCurrentGCSPtr(GetCurrentGCSPtr() + 8);

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';

BranchTo(target, BranchType_RET, FALSE);
```

RETAA, RETAB

Return from subroutine, with pointer authentication. This instruction authenticates the address that is held in LR, using SP as the modifier and the specified key, branches to the authenticated address, with a hint that this instruction is a subroutine return.

Key A is used for RETAA. Key B is used for RETAB.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see *Faulting on pointer authentication*.

The authenticated address is not written back to LR.

Integer  
(FEAT\_PAuth)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	0	1	0	1	1	1	1	1	0	0	0	0	1	M	1	1	1	1	1	1	1	1	1	1
Z								op								A				Rn				Rm							

RETAA (M == 0)

RETAA

RETAB (M == 1)

RETAB

```
boolean use_key_a = (M == '0');  
  
if !HavePACExt() then  
    UNDEFINED;
```

Operation

```
GCSInstruction inst_type;  
bits(64) target = X[30, 64];  
  
bits(64) modifier = SP[];  
  
if use_key_a then  
    target = AuthIA(target, modifier, TRUE);  
else  
    target = AuthIB(target, modifier, TRUE);  
  
if HaveGCS() && GCSPCREnabled(PSTATE.EL) then  
    if use_key_a then  
        inst_type = GCSInstType_PRETAA;  
    else  
        inst_type = GCSInstType_PRETAB;  
        target = LoadCheckGCSRecord(target, inst_type);  
        SetCurrentGCSPtr(GetCurrentGCSPtr() + 8);  
  
// Value in BTypeNext will be used to set PSTATE.BTYPE  
BTypeNext = '00';  
  
BranchTo(target, BranchType_RET, FALSE);
```



# REV

Reverse Bytes reverses the byte order in a register.

This instruction is used by the pseudo-instruction [REV64](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	x	Rn				Rd					
																						opc									

32-bit (sf == 0 && opc == 10)

REV <Wd>, <Wn>

64-bit (sf == 1 && opc == 11)

REV <Xd>, <Xn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
  when '00'
    Unreachable();
  when '01'
    container_size = 16;
  when '10'
    container_size = 32;
  when '11'
    if sf == '0' then UNDEFINED;
    container_size = 64;
```

## Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

## Operation

```
bits(datasize) operand = X[n, datasize];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
  rev_index = index + ((elements_per_container - 1) * 8);
  for e = 0 to elements_per_container-1
    result<rev_index+7:rev_index> = operand<index+7:index>;
    index = index + 8;
    rev_index = rev_index - 8;
X[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

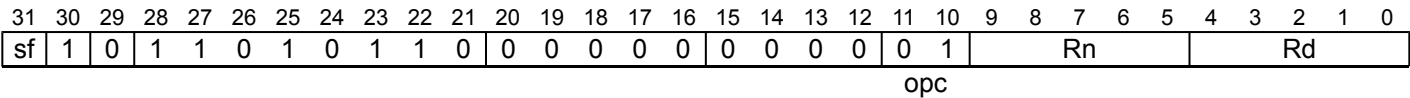
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

REV16

Reverse bytes in 16-bit halfwords reverses the byte order in each 16-bit halfword of a register.



32-bit (sf == 0)

```
REV16 <Wd>, <Wn>
```

64-bit (sf == 1)

```
REV16 <Xd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
  when '00'
    Unreachable();
  when '01'
    container_size = 16;
  when '10'
    container_size = 32;
  when '11'
    if sf == '0' then UNDEFINED;
    container_size = 64;
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n, datasize];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
  rev_index = index + ((elements_per_container - 1) * 8);
  for e = 0 to elements_per_container-1
    result<rev_index+7:rev_index> = operand<index+7:index>;
    index = index + 8;
    rev_index = rev_index - 8;

X[d, datasize] = result;
```

Operational information

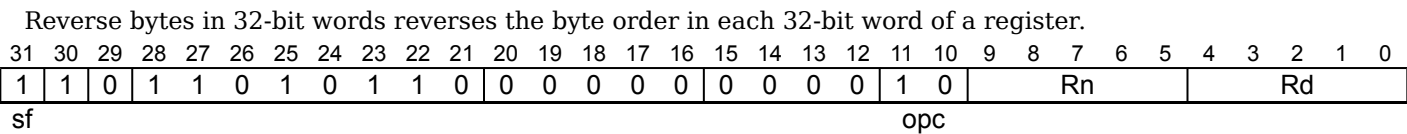
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

REV32



```
REV32 <Xd>, <Xn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
  when '00'
    Unreachable();
  when '01'
    container_size = 16;
  when '10'
    container_size = 32;
  when '11'
    if sf == '0' then UNDEFINED;
    container_size = 64;
```

Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n, datasize];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
  rev_index = index + ((elements_per_container - 1) * 8);
  for e = 0 to elements_per_container-1
    result<rev_index+7:rev_index> = operand<index+7:index>;
    index = index + 8;
    rev_index = rev_index - 8;

X[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

REV64

Reverse Bytes reverses the byte order in a 64-bit general-purpose register.  
When assembling for Armv8.2, an assembler must support this pseudo-instruction. It is OPTIONAL whether an assembler supports this pseudo-instruction when assembling for an architecture earlier than Armv8.2.

This is a pseudo-instruction of REV. This means:

- The encodings in this description are named to match the encodings of REV.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of REV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	Rn				Rd					
sf											opc																				

64-bit

REV64 <Xd>, <Xn>

is equivalent to

REV <Xd>, <Xn>

Assembler Symbols

- <Xd>
- Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>
- Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of REV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

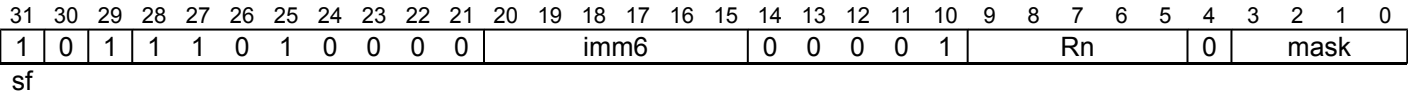
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RMIF

Performs a rotation right of a value held in a general purpose register by an immediate value, and then inserts a selection of the bottom four bits of the result of the rotation into the PSTATE flags, under the control of a second immediate mask.

Integer  
(FEAT\_FlagM)



RMIF <Xn>, #<shift>, #<mask>

```
if !HaveFlagManipulateExt() then UNDEFINED;
integer lsb = UInt(imm6);
integer n = UInt(Rn);
```

Assembler Symbols

- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <shift> Is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
- <mask> Is the flag bit mask, an immediate in the range 0 to 15, which selects the bits that are inserted into the NZCV condition flags, encoded in the "mask" field.

Operation

```
bits(4) tmp;
bits(64) tmpreg = X[n, 64];
tmp = (tmpreg:tmpreg)<lsb+3:lsb>;
if mask<3> == '1' then PSTATE.N = tmp<3>;
if mask<2> == '1' then PSTATE.Z = tmp<2>;
if mask<1> == '1' then PSTATE.C = tmp<1>;
if mask<0> == '1' then PSTATE.V = tmp<0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

ROR (immediate)

Rotate right (immediate) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This is an alias of [EXTR](#). This means:

- The encodings in this description are named to match the encodings of [EXTR](#).
- The description of [EXTR](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	0	0	1	1	1	N	0	Rm				imms				Rn				Rd								

32-bit (sf == 0 && N == 0 && imms == 0xxxxx)

```
ROR <Wd>, <Ws>, #<shift>
```

is equivalent to

```
EXTR <Wd>, <Ws>, <Ws>, #<shift>
```

and is the preferred disassembly when `Rn == Rm`.

64-bit (sf == 1 && N == 1)

```
ROR <Xd>, <Xs>, #<shift>
```

is equivalent to

```
EXTR <Xd>, <Xs>, <Xs>, #<shift>
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Ws>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xs>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<shift>	For the 32-bit variant: is the amount by which to rotate, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the amount by which to rotate, in the range 0 to 63, encoded in the "imms" field.

Operation

The description of [EXTR](#) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.





ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This is an alias of [RORV](#). This means:

- The encodings in this description are named to match the encodings of [RORV](#).
  - The description of [RORV](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.
- |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |     |    |   |   |   |   |  |  |  |  |  |  |  |  |  |  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|-----|----|---|---|---|---|--|--|--|--|--|--|--|--|--|--|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8 | 7 | 6 | 5   | 4  | 3 | 2 | 1 | 0 |  |  |  |  |  |  |  |  |  |  |
| sf | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | Rm |    |    |    |    | 0  | 0  | 1  | 0  | 1  | 1  | Rn |   |   |   |     | Rd |   |   |   |   |  |  |  |  |  |  |  |  |  |  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   | op2 |    |   |   |   |   |  |  |  |  |  |  |  |  |  |  |

32-bit (sf == 0)

ROR <Wd>, <Wn>, <Wm>

is equivalent to

[RORV](#) <Wd>, <Wn>, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

ROR <Xd>, <Xn>, <Xm>

is equivalent to

[RORV](#) <Xd>, <Xn>, <Xm>

and is always the preferred disassembly.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of [RORV](#) gives the operational pseudocode for this instruction.

Operational information

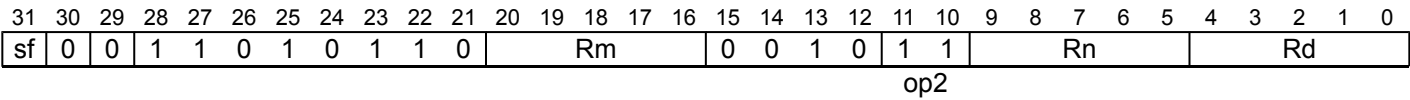
- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



RORV

Rotate Right Variable provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias [ROR \(register\)](#).



32-bit (sf == 0)

```
RORV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
RORV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m, datasize];

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);
X[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

RPRFM

Range Prefetch Memory signals the memory system that data memory accesses from a specified range of addresses are likely to occur in the near future. The instruction may also signal the memory system about the likelihood of data reuse of the specified range of addresses. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as prefetching locations within the specified address ranges into one or more caches. The memory system may also exploit the data reuse hints to decide whether to retain the data in other caches upon eviction from the innermost caches or to discard it.

The effect of an RPRFM instruction is IMPLEMENTATION DEFINED, but because these signals are only hints, the instruction cannot cause a synchronous Data Abort exception and is guaranteed not to access Device memory. It is valid for the PE to treat this instruction as a NOP.

An RPRFM instruction specifies the type of accesses and range of addresses using the following parameters:

- 'Type', in the <rprfop> operand opcode bits, specifies whether the prefetched data will be accessed by load or store instructions.
- 'Policy', in the <rprfop> operand opcode bits, specifies whether the data is likely to be reused or if it is a streaming, non-temporal prefetch. If a streaming prefetch is specified, then the 'ReuseDistance' parameter is ignored.
- 'BaseAddress', in the 64-bit base register, holds the initial block address for the accesses.
- 'ReuseDistance', in the metadata register bits[63:60], indicates the maximum number of bytes to be accessed by this PE before executing the next RPRFM instruction that specifies the same range. This includes the total number of bytes inside and outside of the range that will be accessed by the same PE. This parameter can be used to influence cache eviction and replacement policies, in order to retain the data in the most optimal levels of the memory hierarchy after each access. If software cannot easily determine the amount of other memory that will be accessed, these bits can be set to zero to indicate that 'ReuseDistance' is not known. Otherwise, these four bits encode decreasing powers of two in the range 512MiB (0b0001) to 32KiB (0b1111).
- 'Stride', in the metadata register bits[59:38], is a signed, two's complement integer encoding of the number of bytes to advance the block address after 'Length' bytes have been accessed, in the range -2MiB to +2MiB-1B. A negative value indicates that the block address is advanced in a descending direction.
- 'Count', in the metadata register bits[37:22], is an unsigned integer encoding of the number of blocks of data to be accessed minus 1, representing the range 1 to 65536 blocks. If 'Count' is 0, then the 'Stride' parameter is ignored and only a single block of contiguous bytes from 'BaseAddress' to ('BaseAddress' + 'Length' - 1) is described.
- 'Length', in the metadata register bits[21:0], is a signed, two's complement integer encoding of the number of contiguous bytes to be accessed starting from the current block address, without changing the block address, in the range -2MiB to +2MiB-1B. A negative value indicates that the bytes are accessed in a descending direction.

Note

Software is expected to honor the parameters it provides to the RPRFM instruction, and the same PE should access all locations in the range, in the direction specified by the sign of the 'Length' and 'Stride' parameters. A range prefetch is considered active on a PE until all locations in the range have been accessed by the PE. A range prefetch might also be inactivated by the PE prior to completion, for example due to a software context switch or lack of hardware resources.

Software should not specify overlapping addresses in multiple active ranges. If a range is expected to be accessed by both load and store instructions (read-modify-write), then a single range with a 'Type' parameter of PST (prefetch for store) should be specified.

Integer  
(FEAT\_RPRFM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	0	0	0	1	0	1	Rm				x	1	x	S	1	0	Rn				1	1	x	x	x			
size				opc							option										Rt											

```
RPRFM (<rprfop>|<#imm6>), <Xm>, [<Xn|SP>]
```

```
bits(6) operation = option<2>:option<0>:S:Rt<2:0>;
integer n = UInt(Rn);
integer m = UInt(Rm);
```

## Assembler Symbols

- <rprfop>** Is the range prefetch operation, defined as <type><policy>. <type> is one of:
- PLD**  
Prefetch for load, encoded in the "Rt<0>" field as 0.
  - PST**  
Prefetch for store, encoded in the "Rt<0>" field as 1.
- <policy> is one of:
- KEEP**  
Retained or temporal prefetch, for data that is expected to be kept in caches to be accessed more than once, encoded in the "option<2>:option<0>:S:Rt<2:1>" fields as 0b00000.
  - STRM**  
Streaming or non-temporal prefetch, for data that is expected to be accessed once and not reused, encoded in the "option<2>:option<0>:S:Rt<2:1>" fields as 0b00010.
- For other encodings of the "option<2>:option<0>:S:Rt<2:0>" fields, use <imm6>.
- <imm6>** Is the range prefetch operation encoding as an immediate, in the range 0 to 63, encoded in "option<2>:option<0>:S:Rt<2:0>". This syntax is only for encodings that are not representable using <rprfop>.
- <Xm>** Is the 64-bit name of the general-purpose register that holds an encoding of the metadata, encoded in the "Rm" field.
- <Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address = if n == 31 then SP[] else X[n, 64];
bits(64) metadata = X[m, 64];
integer stride = SInt(metadata<59:38>);
integer count = UInt(metadata<37:22>) + 1;
integer length = SInt(metadata<21:0>);
integer reuse;

if metadata<63:60> == '0000' then
    reuse = -1;    // Not known
else
    reuse = 32768 << (15 - UInt(metadata<63:60>));

Hint\_RangePrefetch(address, length, stride, count, reuse, operation);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SB

Speculation Barrier is a barrier that controls speculation.

The semantics of the Speculation Barrier are that the execution, until the barrier completes, of any instruction that appears later in the program order than the barrier:

- Cannot be performed speculatively to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception.

In particular, any instruction that appears later in the program order than the barrier cannot cause a speculative allocation into any caching structure where the allocation of that entry could be indicative of any data value present in memory or in the registers.

The SB instruction:

- Cannot be speculatively executed as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception. The potentially exception generating instruction can complete once it is known not to be speculative, and all data values generated by instructions appearing in program order before the SB instruction have their predicted values confirmed.

When the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after an uncompleted SB instruction, the SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched.

System  
(FEAT\_SB)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	(0)	(0)	(0)	(0)	1	1	1	1	1	1	1	1			
																				CRm				opc										

SB

```
if !HaveSBExt() then UNDEFINED;
```

Operation

```
SpeculationBarrier();
```

SBC

Subtract with Carry subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

This instruction is used by the alias [NGC](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	0	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0	Rn					Rd				
op S																															

32-bit (sf == 0)

SBC <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

SBC <Xd>, <Xn>, <Xm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Alias Conditions

Alias	Is preferred when
<a href="#">NGC</a>	Rn == '11111'

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];

operand2 = NOT(operand2);

(result, -) = AddWithCarry(operand1, operand2, PSTATE.C);

X[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.





SBCS

Subtract with Carry, setting flags, subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias [NGCS](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	1	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0	Rn					Rd				
op S																															

32-bit (sf == 0)

SBCS <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

SBCS <Xd>, <Xn>, <Xm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Alias Conditions

Alias	Is preferred when
<a href="#">NGCS</a>	Rn == '11111'

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
bits(4) nzcvc;

operand2 = NOT(operand2);

(result, nzcvc) = AddWithCarry(operand1, operand2, PSTATE.C);

PSTATE.<N,Z,C,V> = nzcvc;

X[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SBFIZ

Signed Bitfield Insert in Zeros copies a bitfield of <width> bits from the least significant bits of the source register to bit position <lsb> of the destination register, setting the destination bits below the bitfield to zero, and the bits above the bitfield to a copy of the most significant bit of the bitfield.

This is an alias of [SBFM](#). This means:

- The encodings in this description are named to match the encodings of [SBFM](#).
- The description of [SBFM](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
sf	0	0	1	0	0	1	1	0	N	immr						imms						Rn						Rd					
opc																																	

32-bit (sf == 0 && N == 0)

SBFIZ <Wd>, <Wn>, #<lsb>, #<width>

is equivalent to

SBFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

64-bit (sf == 1 && N == 1)

SBFIZ <Xd>, <Xn>, #<lsb>, #<width>

is equivalent to

SBFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <lsb> For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.  
For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
- <width> For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.  
For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of [SBFM](#) gives the operational pseudocode for this instruction.

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



SBFM

Signed Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If <imms> is greater than or equal to <immr>, this copies a bitfield of (<imms>-<immr>+1) bits starting from bit position <immr> in the source register to the least significant bits of the destination register.

If <imms> is less than <immr>, this copies a bitfield of (<imms>+1) bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases the destination bits below the bitfield are set to zero, and the bits above the bitfield are set to a copy of the most significant bit of the bitfield.

This instruction is used by the aliases [ASR \(immediate\)](#), [SBFIZ](#), [SBFX](#), [SXTB](#), [SXTH](#), and [SXTW](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		0	0	1	0	0	1	1	0	N	immr					imms					Rn					Rd					
opc																															

32-bit (sf == 0 && N == 0)

SBFM <Wd>, <Wn>, #<immr>, #<imms>

64-bit (sf == 1 && N == 1)

SBFM <Xd>, <Xn>, #<immr>, #<imms>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

integer r;
integer s;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;

r = UInt(immr);
s = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE, datasize);
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <immr> For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.  
For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.
- <imms> For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.  
For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

Alias	Of variant	Is preferred when
<a href="#">ASR (immediate)</a>	32-bit	imms == '011111'
<a href="#">ASR (immediate)</a>	64-bit	imms == '111111'

Alias	Of variant	Is preferred when
<a href="#">SBFIZ</a>		<a href="#">UInt</a> (imms) < <a href="#">UInt</a> (immr)
<a href="#">SBFX</a>		<a href="#">BFXPreferred</a> (sf, opc<1>, imms, immr)
<a href="#">SXTB</a>		immr == '000000' && imms == '000111'
<a href="#">SXTH</a>		immr == '000000' && imms == '001111'
<a href="#">SXTW</a>		immr == '000000' && imms == '011111'

## Operation

```
bits(datasize) src = X[n, datasize];

// perform bitfield move on low bits
bits(datasize) bot = ROR(src, r) AND wmask;

// determine extension bits (sign, zero or dest register)
bits(datasize) top = Replicate(src<s>, datasize);

// combine extension bits and result bits
X[d, datasize] = (top AND NOT(tmask)) OR (bot AND tmask);
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SBFX

Signed Bitfield Extract copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to a copy of the most significant bit of the bitfield.

This is an alias of [SBFM](#). This means:

- The encodings in this description are named to match the encodings of [SBFM](#).
- The description of [SBFM](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	0	0	1	1	0	N	immr						imms						Rn				Rd					

opc

32-bit (sf == 0 && N == 0)

SBFX <Wd>, <Wn>, #<lsb>, #<width>

is equivalent to

SBFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

64-bit (sf == 1 && N == 1)

SBFX <Xd>, <Xn>, #<lsb>, #<width>

is equivalent to

SBFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
<width>	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of [SBFM](#) gives the operational pseudocode for this instruction.

Operational information

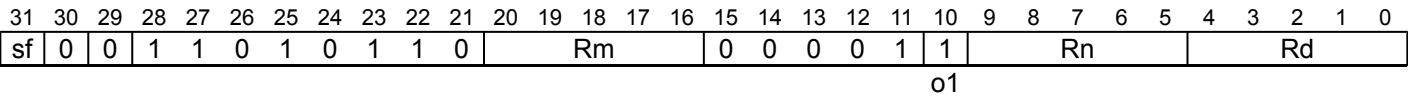
- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.





SDIV

Signed Divide divides a signed integer register value by another signed integer register value, and writes the result to the destination register. The condition flags are not affected.



32-bit (sf == 0)

```
SDIV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
SDIV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
integer result;

if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, FALSE)) / Real(Int(operand2, FALSE)));

X[d, datasize] = result<datasize-1:0>;
```

## SETF8, SETF16

Set the PSTATE.NZV flags based on the value in the specified general-purpose register. SETF8 treats the value as an 8 bit value, and SETF16 treats the value as an 16 bit value.  
The PSTATE.C flag is not affected by these instructions.

### Integer (FEAT\_FlagM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	sz	0	0	1	0	Rn					0	1	1	0	1
sf																															

### SETF8 (sz == 0)

SETF8 <Wn>

### SETF16 (sz == 1)

SETF16 <Wn>

```
if !HaveFlagManipulateExt() then UNDEFINED;
integer msb = if sz == '1' then 15 else 7;
integer n = UInt(Rn);
```

### Assembler Symbols

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

### Operation

```
bits(32) tmpreg = X[n, 32];
PSTATE.N = tmpreg<msb>;
PSTATE.Z = if (tmpreg<msb:0> == Zeros(msb + 1)) then '1' else '0';
PSTATE.V = tmpreg<msb+1> EOR tmpreg<msb>;
//PSTATE.C unchanged;
```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

## SETGP, SETGM, SETGE

Memory Set with tag setting. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGP, then SETGM, and then SETGE.

SETGP performs some preconditioning of the arguments suitable for using the SETGM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGM performs an IMPLEMENTATION DEFINED amount of the memory set. SETGE performs the last part of the memory set.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGP, option A (which results in encoding PSTATE.C = 0):

- If  $Xn_{<63>} == 1$ , the set size is saturated to  $0x7FFFFFFFFFFFFFFF0$ .
- $Xd$  holds the original  $Xd$  + saturated  $Xn$ .
- $Xn$  holds  $-1 * \text{saturated } Xn + \text{an IMPLEMENTATION DEFINED number of bytes set}$ .
- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .

After execution of SETGP, option B (which results in encoding PSTATE.C = 1):

- If  $Xn_{<63>} == 1$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF0$ .
- $Xd$  holds the original  $Xd$  + an IMPLEMENTATION DEFINED number of bytes set.
- $Xn$  holds the saturated  $Xn$  - an IMPLEMENTATION DEFINED number of bytes set.
- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .

For SETGM, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $Xn$  is treated as a signed 64-bit number.
- $Xn$  holds  $-1 * \text{number of bytes remaining to be set in the memory set in total}$ .
- $Xd$  holds the lowest address that the set is made to  $-Xn$ .
- At the end of the instruction, the value of  $Xn$  is written back with  $-1 * \text{number of bytes remaining to be set in the memory set in total}$ .

For SETGM, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $Xn$  holds the number of bytes remaining to be set in the memory set in total.
- $Xd$  holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of  $Xn$  is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of  $Xd$  is written back with the lowest address that has not been set.

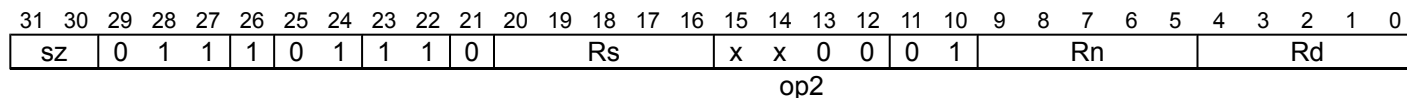
For SETGE, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $Xn$  is treated as a signed 64-bit number.
- $Xn$  holds  $-1 * \text{the number of bytes remaining to be set in the memory set in total}$ .
- $Xd$  holds the lowest address that the set is made to  $-Xn$ .
- At the end of the instruction, the value of  $Xn$  is written back with 0.

For SETGE, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $Xn$  holds the number of bytes remaining to be set in the memory set in total.
- $Xd$  holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of  $Xn$  is written back with 0.
  - the value of  $Xd$  is written back with the lowest address that has not been set.

## Integer (FEAT\_MOPS)



### Epilogue (op2 == 1000)

SETGE [<Xd>]!, <Xn>!, <Xs>

### Main (op2 == 0100)

SETGM [<Xd>]!, <Xn>!, <Xs>

### Prologue (op2 == 0000)

SETGP [<Xd>]!, <Xn>!, <Xs>

```

if !HaveFeatMOPS() || !HaveMTEExt() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;
boolean nontemporal = options<1> == '1';

MOPSSStage stage;
case op2<3:2> of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise UNDEFINED;

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set SET\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.
- <Xs> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.
- For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.



```

bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
boolean is_setg = TRUE;
integer B;

boolean implements_option_a = SETGOptionA();
boolean privileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescSTGMOPS(privileged, nontemporal);

if stage == MOPSSStage Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFFF0<63:0>;

    if ((!IsZero(setsize) && !IsAligned(toaddress, TAG_GRANULE)) || !IsAligned(setsize, TAG_GRANULE)) then
        AArch64.Abort(toaddress, AlignmentFault(accdesc));

    if implements_option_a then
        nzcvc = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcvc = '0010';
else
    CheckMemSetParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, setsize, is_setg);

    if ((!IsZero(setsize) && !IsAligned(toaddress, TAG_GRANULE)) || !IsAligned(setsize, TAG_GRANULE)) then
        AArch64.Abort(toaddress, AlignmentFault(accdesc));

bits(64) stagesetsize = MemSetStageSize(stage, toaddress, setsize, is_setg);

integer tagstep;
bits(4) tag;
bits(64) tagaddr;

if implements_option_a then
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= -1 * SInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress+setsize, B, accdesc] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress + setsize);
        while tagstep > 0 do
            tagaddr = toaddress + setsize + (tagstep - 1) * 16;
            AArch64.MemTag[tagaddr, accdesc] = tag;
            tagstep = tagstep - 1;

        setsize = setsize + B;
        stagesetsize = stagesetsize + B;

        if stage != MOPSSStage Prologue then
            X[n, 64] = setsize;
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= UInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress, B, accdesc] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress);

```

```

while tagstep > 0 do
    tagaddr = toaddress + (tagstep - 1) * 16;
    AArch64.MemTag[tagaddr, accdesc] = tag;
    tagstep = tagstep - 1;

toaddress = toaddress + B;
setsize = setsize - B;
stagesetsize = stagesetsize - B;

if stage != MOPSStage\_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;

if stage == MOPSStage\_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;
    PSTATE.<N,Z,C,V> = nzcvc;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## SETGPN, SETGMN, SETGEN

Memory Set with tag setting, non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPN, then SETGMN, and then SETGEN.

SETGPN performs some preconditioning of the arguments suitable for using the SETGMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGMN performs an IMPLEMENTATION DEFINED amount of the memory set. SETGEN performs the last part of the memory set.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGPN, option A (which results in encoding PSTATE.C = 0):

- If  $Xn < 63 > == 1$ , the set size is saturated to  $0x7FFFFFFFFFFFFFFF0$ .
- Xd holds the original Xd + saturated Xn.
- Xn holds  $-1 * \text{saturated Xn} + \text{an IMPLEMENTATION DEFINED number of bytes set}$ .
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETGPN, option B (which results in encoding PSTATE.C = 1):

- If  $Xn < 63 > == 1$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF0$ .
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETGMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds  $-1 * \text{number of bytes remaining to be set in the memory set in total}$ .
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with  $-1 * \text{number of bytes remaining to be set in the memory set in total}$ .

For SETGMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

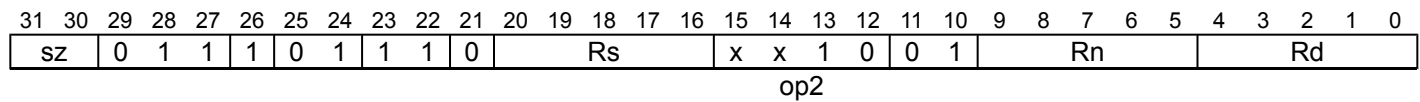
For SETGEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds  $-1 * \text{the number of bytes remaining to be set in the memory set in total}$ .
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETGEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

**Integer**  
(FEAT\_MOPS)



### Epilogue (op2 == 1010)

SETGEN [<Xd>]!, <Xn>!, <Xs>

### Main (op2 == 0110)

SETGMN [<Xd>]!, <Xn>!, <Xs>

### Prologue (op2 == 0010)

SETGPN [<Xd>]!, <Xn>!, <Xs>

```

if !HaveFeatMOPS() || !HaveMTEExt() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;
boolean nontemporal = options<1> == '1';

MOPSSStage stage;
case op2<3:2> of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise UNDEFINED;

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set SET\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.
- <Xs> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.
- For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.



```

bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
boolean is_setg = TRUE;
integer B;

boolean implements_option_a = SETGOptionA();
boolean privileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescSTGMOPS(privileged, nontemporal);

if stage == MOPSSStage Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFFF0<63:0>;

    if ((!IsZero(setsize) && !IsAligned(toaddress, TAG_GRANULE)) || !IsAligned(setsize, TAG_GRANULE)) then
        AArch64.Abort(toaddress, AlignmentFault(accdesc));

    if implements_option_a then
        nzcvc = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcvc = '0010';
else
    CheckMemSetParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, setsize, is_setg);

    if ((!IsZero(setsize) && !IsAligned(toaddress, TAG_GRANULE)) || !IsAligned(setsize, TAG_GRANULE)) then
        AArch64.Abort(toaddress, AlignmentFault(accdesc));

bits(64) stagesetsize = MemSetStageSize(stage, toaddress, setsize, is_setg);

integer tagstep;
bits(4) tag;
bits(64) tagaddr;

if implements_option_a then
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= -1 * SInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress+setsize, B, accdesc] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress + setsize);
        while tagstep > 0 do
            tagaddr = toaddress + setsize + (tagstep - 1) * 16;
            AArch64.MemTag[tagaddr, accdesc] = tag;
            tagstep = tagstep - 1;

        setsize = setsize + B;
        stagesetsize = stagesetsize + B;

        if stage != MOPSSStage Prologue then
            X[n, 64] = setsize;
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= UInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress, B, accdesc] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress);

```

```

while tagstep > 0 do
    tagaddr = toaddress + (tagstep - 1) * 16;
    AArch64.MemTag[tagaddr, accdesc] = tag;
    tagstep = tagstep - 1;

toaddress = toaddress + B;
setsize = setsize - B;
stagesetsize = stagesetsize - B;

if stage != MOPSStage\_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;

if stage == MOPSStage\_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;
    PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SETGPT, SETGMT, SETGET

Memory Set with tag setting, unprivileged. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPT, then SETGMT, and then SETGET.

SETGPT performs some preconditioning of the arguments suitable for using the SETGMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGMT performs an IMPLEMENTATION DEFINED amount of the memory set. SETGET performs the last part of the memory set.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGPT, option A (which results in encoding PSTATE.C = 0):

- If  $Xn < 63 > == 1$ , the set size is saturated to  $0x7FFFFFFFFFFFFFFF0$ .
- $Xd$  holds the original  $Xd +$  saturated  $Xn$ .
- $Xn$  holds  $-1 * \text{saturated } Xn + \text{an IMPLEMENTATION DEFINED number of bytes set}$ .
- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .

After execution of SETGPT, option B (which results in encoding PSTATE.C = 1):

- If  $Xn < 63 > == 1$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF0$ .
- $Xd$  holds the original  $Xd +$  an IMPLEMENTATION DEFINED number of bytes set.
- $Xn$  holds the saturated  $Xn -$  an IMPLEMENTATION DEFINED number of bytes set.
- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .

For SETGMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $Xn$  is treated as a signed 64-bit number.
- $Xn$  holds  $-1 * \text{number of bytes remaining to be set in the memory set in total}$ .
- $Xd$  holds the lowest address that the set is made to  $-Xn$ .
- At the end of the instruction, the value of  $Xn$  is written back with  $-1 * \text{number of bytes remaining to be set in the memory set in total}$ .

For SETGMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $Xn$  holds the number of bytes remaining to be set in the memory set in total.
- $Xd$  holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of  $Xn$  is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of  $Xd$  is written back with the lowest address that has not been set.

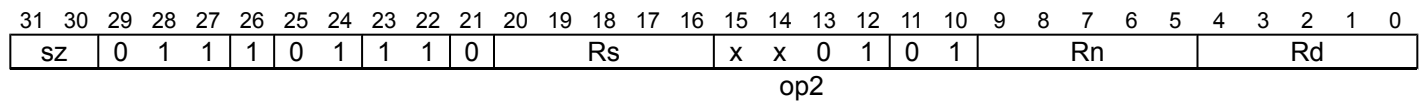
For SETGET, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $Xn$  is treated as a signed 64-bit number.
- $Xn$  holds  $-1 * \text{the number of bytes remaining to be set in the memory set in total}$ .
- $Xd$  holds the lowest address that the set is made to  $-Xn$ .
- At the end of the instruction, the value of  $Xn$  is written back with 0.

For SETGET, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $Xn$  holds the number of bytes remaining to be set in the memory set in total.
- $Xd$  holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of  $Xn$  is written back with 0.
  - the value of  $Xd$  is written back with the lowest address that has not been set.

**Integer**  
(FEAT\_MOPS)



### Epilogue (op2 == 1001)

SETGET [<Xd>]!, <Xn>!, <Xs>

### Main (op2 == 0101)

SETGMT [<Xd>]!, <Xn>!, <Xs>

### Prologue (op2 == 0001)

SETGPT [<Xd>]!, <Xn>!, <Xs>

```

if !HaveFeatMOPS() || !HaveMTEExt() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;
boolean nontemporal = options<1> == '1';

MOPSSStage stage;
case op2<3:2> of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise UNDEFINED;

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set SET\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.
- <Xs> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.
- For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.





```

bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
boolean is_setg = TRUE;
integer B;

boolean implements_option_a = SETGOptionA();
boolean privileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescSTGMOPS(privileged, nontemporal);

if stage == MOPSSStage Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFFF0<63:0>;

    if ((!IsZero(setsize) && !IsAligned(toaddress, TAG_GRANULE)) || !IsAligned(setsize, TAG_GRANULE)) then
        AArch64.Abort(toaddress, AlignmentFault(accdesc));

    if implements_option_a then
        nzcvc = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcvc = '0010';
else
    CheckMemSetParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, setsize, is_setg);

    if ((!IsZero(setsize) && !IsAligned(toaddress, TAG_GRANULE)) || !IsAligned(setsize, TAG_GRANULE)) then
        AArch64.Abort(toaddress, AlignmentFault(accdesc));

bits(64) stagesetsize = MemSetStageSize(stage, toaddress, setsize, is_setg);

integer tagstep;
bits(4) tag;
bits(64) tagaddr;

if implements_option_a then
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= -1 * SInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress+setsize, B, accdesc] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress + setsize);
        while tagstep > 0 do
            tagaddr = toaddress + setsize + (tagstep - 1) * 16;
            AArch64.MemTag[tagaddr, accdesc] = tag;
            tagstep = tagstep - 1;

        setsize = setsize + B;
        stagesetsize = stagesetsize + B;

        if stage != MOPSSStage Prologue then
            X[n, 64] = setsize;
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= UInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress, B, accdesc] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress);

```

```

while tagstep > 0 do
    tagaddr = toaddress + (tagstep - 1) * 16;
    AArch64.MemTag[tagaddr, accdesc] = tag;
    tagstep = tagstep - 1;

toaddress = toaddress + B;
setsize = setsize - B;
stagesetsize = stagesetsize - B;

if stage != MOPSStage\_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;

if stage == MOPSStage\_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;
    PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SETGPTN, SETGMTN, SETGETN

Memory Set with tag setting, unprivileged and non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPTN, then SETGMTN, and then SETGETN.

SETGPTN performs some preconditioning of the arguments suitable for using the SETGMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGMTN performs an IMPLEMENTATION DEFINED amount of the memory set. SETGETN performs the last part of the memory set.

### Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

### Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGPTN, option A (which results in encoding PSTATE.C = 0):

- If  $Xn < 63 > == 1$ , the set size is saturated to  $0x7FFFFFFFFFFFFFFF0$ .
- Xd holds the original Xd + saturated Xn.
- Xn holds  $-1 * \text{saturated Xn} + \text{an IMPLEMENTATION DEFINED number of bytes set}$ .
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETGPTN, option B (which results in encoding PSTATE.C = 1):

- If  $Xn < 63 > == 1$ , the copy size is saturated to  $0x7FFFFFFFFFFFFFFF0$ .
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETGMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds  $-1 * \text{number of bytes remaining to be set in the memory set in total}$ .
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with  $-1 * \text{number of bytes remaining to be set in the memory set in total}$ .

For SETGMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

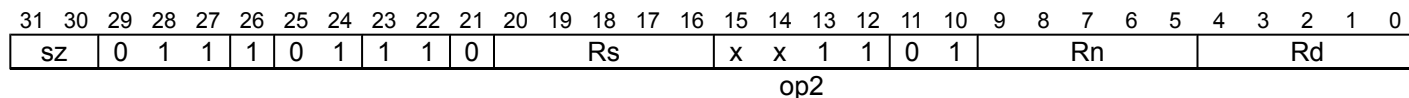
For SETGETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds  $-1 * \text{the number of bytes remaining to be set in the memory set in total}$ .
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETGETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

**Integer**  
(FEAT\_MOPS)



### Epilogue (op2 == 1011)

SETGETN [<Xd>]!, <Xn>!, <Xs>

### Main (op2 == 0111)

SETGMTN [<Xd>]!, <Xn>!, <Xs>

### Prologue (op2 == 0011)

SETGPTN [<Xd>]!, <Xn>!, <Xs>

```

if !HaveFeatMOPS() || !HaveMTEExt() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;
boolean nontemporal = options<1> == '1';

MOPSSStage stage;
case op2<3:2> of
  when '00' stage = MOPSSStage Prologue;
  when '01' stage = MOPSSStage Main;
  when '10' stage = MOPSSStage Epilogue;
  otherwise UNDEFINED;

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set SET\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.
- <Xs> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.
- For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.



```

bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
boolean is_setg = TRUE;
integer B;

boolean implements_option_a = SETGOptionA();
boolean privileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescSTGMOPS(privileged, nontemporal);

if stage == MOPSSStage Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFFF0<63:0>;

    if ((!IsZero(setsize) && !IsAligned(toaddress, TAG_GRANULE)) || !IsAligned(setsize, TAG_GRANULE)) then
        AArch64.Abort(toaddress, AlignmentFault(accdesc));

    if implements_option_a then
        nzcvc = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcvc = '0010';
else
    CheckMemSetParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, setsize, is_setg);

    if ((!IsZero(setsize) && !IsAligned(toaddress, TAG_GRANULE)) || !IsAligned(setsize, TAG_GRANULE)) then
        AArch64.Abort(toaddress, AlignmentFault(accdesc));

bits(64) stagesetsize = MemSetStageSize(stage, toaddress, setsize, is_setg);

integer tagstep;
bits(4) tag;
bits(64) tagaddr;

if implements_option_a then
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= -1 * SInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress+setsize, B, accdesc] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress + setsize);
        while tagstep > 0 do
            tagaddr = toaddress + setsize + (tagstep - 1) * 16;
            AArch64.MemTag[tagaddr, accdesc] = tag;
            tagstep = tagstep - 1;

        setsize = setsize + B;
        stagesetsize = stagesetsize + B;

        if stage != MOPSSStage Prologue then
            X[n, 64] = setsize;
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= UInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress, B, accdesc] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress);

```

```

while tagstep > 0 do
    tagaddr = toaddress + (tagstep - 1) * 16;
    AArch64.MemTag[tagaddr, accdesc] = tag;
    tagstep = tagstep - 1;

toaddress = toaddress + B;
setsize = setsize - B;
stagesetsize = stagesetsize - B;

if stage != MOPSStage\_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;

if stage == MOPSStage\_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;
    PSTATE.<N,Z,C,V> = nzcv;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SETP, SETM, SETE

Memory Set. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETP, then SETM, and then SETE.

SETP performs some preconditioning of the arguments suitable for using the SETM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETM performs an IMPLEMENTATION DEFINED amount of the memory set. SETE performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETP, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1\* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETP, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETM, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1\* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1\* the number of bytes remaining to be set in the memory set in total.

For SETM, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETE, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1\* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETE, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer
(FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sz		0	1	1	0	0	1	1	1	0	Rs					x	x	0	0	0	1	Rn					Rd				
op2																															



### Epilogue (op2 == 1000)

```
SETE [<Xd>]!, <Xn>!, <Xs>
```

### Main (op2 == 0100)

```
SETM [<Xd>]!, <Xn>!, <Xs>
```

### Prologue (op2 == 0000)

```
SETP [<Xd>]!, <Xn>!, <Xs>
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;
boolean nontemporal = options<1> == '1';

MOPSStage stage;
case op2<3:2> of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise UNDEFINED;

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [Memory Copy and Memory Set SET\\*](#).

### Assembler Symbols

- |                   |   |
|-------------------|---|
| <b>&lt;Xd&gt;</b> | For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.<br><br>For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.   |
| <b>&lt;Xn&gt;</b> | For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field.<br><br>For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.<br><br>For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field. |
| <b>&lt;Xs&gt;</b> | Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.   |

## Operation

```
bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
boolean is_setg = FALSE;
integer B;

boolean implements_option_a = SETOptionA();
boolean privileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescMOPS(MemOp_STORE, privileged, nontemporal);

if stage == MOPSSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcvc = '0010';

else
    CheckMemSetParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, setsize, is_setg);

bits(64) stagesetsize = MemSetStageSize(stage, toaddress, setsize, is_setg);

if implements_option_a then
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 1);
        assert B <= -1 * SInt(stagesetsize);

        Mem[toaddress+setsize, B, accdesc] = Replicate(data, B);

        setsize = setsize + B;
        stagesetsize = stagesetsize + B;

        if stage != MOPSSStage_Prologue then
            X[n, 64] = setsize;
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 1);
        assert B <= UInt(stagesetsize);

        Mem[toaddress, B, accdesc] = Replicate(data, B);

        toaddress = toaddress + B;
        setsize = setsize - B;
        stagesetsize = stagesetsize - B;

        if stage != MOPSSStage_Prologue then
            X[n, 64] = setsize;
            X[d, 64] = toaddress;

if stage == MOPSSStage_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;
    PSTATE.<N,Z,C,V> = nzcvc;
```

SETPN, SETMN, SETEN

Memory Set, non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPN, then SETMN, and then SETEN.

SETPN performs some preconditioning of the arguments suitable for using the SETMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETMN performs an IMPLEMENTATION DEFINED amount of the memory set. SETEN performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETPN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1\* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETPN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1\* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1\* the number of bytes remaining to be set in the memory set in total.

For SETMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1\* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer (FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sz		0	1	1	0	0	1	1	1	0	Rs					x	x	1	0	0	1	Rn					Rd				
op2																															

## Epilogue (op2 == 1010)

```
SETEN [<Xd>]!, <Xn>!, <Xs>
```

## Main (op2 == 0110)

```
SETMN [<Xd>]!, <Xn>!, <Xs>
```

## Prologue (op2 == 0010)

```
SETPN [<Xd>]!, <Xn>!, <Xs>
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;
boolean nontemporal = options<1> == '1';

MOPSStage stage;
case op2<3:2> of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise UNDEFINED;

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [Memory Copy and Memory Set SET\\*](#).

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.
- <Xs> Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

## Operation

```
bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
boolean is_setg = FALSE;
integer B;

boolean implements_option_a = SETOptionA();
boolean privileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescMOPS(MemOp\_STORE, privileged, nontemporal);

if stage == MOPSSStage\_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcvc = '0010';

else
    CheckMemSetParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, setsize, is_setg);

bits(64) stagesetsize = MemSetStageSize(stage, toaddress, setsize, is_setg);

if implements_option_a then
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 1);
        assert B <= -1 * SInt(stagesetsize);

        Mem[toaddress+setsize, B, accdesc] = Replicate(data, B);

        setsize = setsize + B;
        stagesetsize = stagesetsize + B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = setsize;
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 1);
        assert B <= UInt(stagesetsize);

        Mem[toaddress, B, accdesc] = Replicate(data, B);

        toaddress = toaddress + B;
        setsize = setsize - B;
        stagesetsize = stagesetsize - B;

        if stage != MOPSSStage\_Prologue then
            X[n, 64] = setsize;
            X[d, 64] = toaddress;

if stage == MOPSSStage\_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;
    PSTATE.<N,Z,C,V> = nzcvc;
```

SETPT, SETMT, SETET

Memory Set, unprivileged. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPT, then SETMT, and then SETET.

SETPT performs some preconditioning of the arguments suitable for using the SETMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETMT performs an IMPLEMENTATION DEFINED amount of the memory set. SETET performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETPT, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1\* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETPT, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1\* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1\* the number of bytes remaining to be set in the memory set in total.

For SETMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

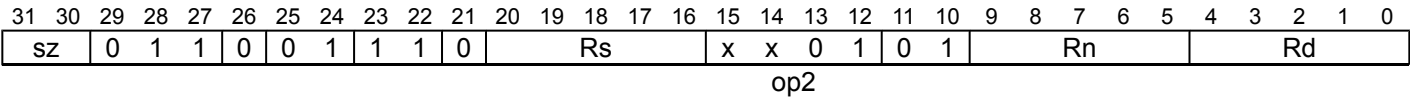
For SETET, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1\* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETET, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer
(FEAT\_MOPS)



## Epilogue (op2 == 1001)

```
SETET [<Xd>]!, <Xn>!, <Xs>
```

## Main (op2 == 0101)

```
SETMT [<Xd>]!, <Xn>!, <Xs>
```

## Prologue (op2 == 0001)

```
SETPT [<Xd>]!, <Xn>!, <Xs>
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;
boolean nontemporal = options<1> == '1';

MOPSStage stage;
case op2<3:2> of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise UNDEFINED;

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [Memory Copy and Memory Set SET\\*](#).

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.
- <Xs> Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

## Operation

```
bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcvc = PSTATE.<N,Z,C,V>;
boolean is_setg = FALSE;
integer B;

boolean implements_option_a = SETOptionA();
boolean privileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescMOPS(MemOp_STORE, privileged, nontemporal);

if stage == MOPSSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcvc = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcvc = '0010';

else
    CheckMemSetParams(stage, implements_option_a, nzcvc, options, d, s, n, toaddress, setsize, is_setg);

bits(64) stagesetsize = MemSetStageSize(stage, toaddress, setsize, is_setg);

if implements_option_a then
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 1);
        assert B <= -1 * SInt(stagesetsize);

        Mem[toaddress+setsize, B, accdesc] = Replicate(data, B);

        setsize = setsize + B;
        stagesetsize = stagesetsize + B;

        if stage != MOPSSStage_Prologue then
            X[n, 64] = setsize;
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 1);
        assert B <= UInt(stagesetsize);

        Mem[toaddress, B, accdesc] = Replicate(data, B);

        toaddress = toaddress + B;
        setsize = setsize - B;
        stagesetsize = stagesetsize - B;

        if stage != MOPSSStage_Prologue then
            X[n, 64] = setsize;
            X[d, 64] = toaddress;

if stage == MOPSSStage_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;
    PSTATE.<N,Z,C,V> = nzcvc;
```



SETPTN, SETMTN, SETETN

Memory Set, unprivileged and non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPTN, then SETMTN, and then SETETN.

SETPTN performs some preconditioning of the arguments suitable for using the SETMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETMTN performs an IMPLEMENTATION DEFINED amount of the memory set. SETETN performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETPTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1\* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETPTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1\* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1\* the number of bytes remaining to be set in the memory set in total.

For SETMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

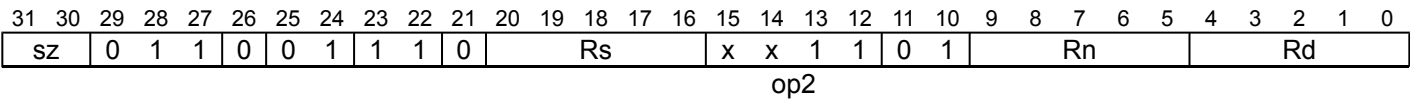
For SETETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1\* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer (FEAT\_MOPS)



### Epilogue (op2 == 1011)

```
SETETN [<Xd>]!, <Xn>!, <Xs>
```

### Main (op2 == 0111)

```
SETMTN [<Xd>]!, <Xn>!, <Xs>
```

### Prologue (op2 == 0011)

```
SETPTN [<Xd>]!, <Xn>!, <Xs>
```

```
if !HaveFeatMOPS() || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;
boolean nontemporal = options<1> == '1';

MOPSStage stage;
case op2<3:2> of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise UNDEFINED;

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [Memory Copy and Memory Set SET\\*](#).

### Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.
- <Xs> Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

## Operation

```
bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
boolean is_setg = FALSE;
integer B;

boolean implements_option_a = SETOptionA();
boolean privileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescMOPS(MemOp_STORE, privileged, nontemporal);

if stage == MOPSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if implements_option_a then
        nzcv = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcv = '0010';

else
    CheckMemSetParams(stage, implements_option_a, nzcv, options, d, s, n, toaddress, setsize, is_setg);

bits(64) stagesetsize = MemSetStageSize(stage, toaddress, setsize, is_setg);

if implements_option_a then
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 1);
        assert B <= -1 * SInt(stagesetsize);

        Mem[toaddress+setsize, B, accdesc] = Replicate(data, B);

        setsize = setsize + B;
        stagesetsize = stagesetsize + B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 1);
        assert B <= UInt(stagesetsize);

        Mem[toaddress, B, accdesc] = Replicate(data, B);

        toaddress = toaddress + B;
        setsize = setsize - B;
        stagesetsize = stagesetsize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
            X[d, 64] = toaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

SEV

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see *Wait for Event mechanism and Send event*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	1	1	1	1	1
																CRm				op2											

SEV

// Empty.

Operation

[SendEvent](#) ();

SEVL

Send Event Local is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop which starts with a `WFE` instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	1	1	1	1	1
																CRm				op2											

SEVL

// Empty.

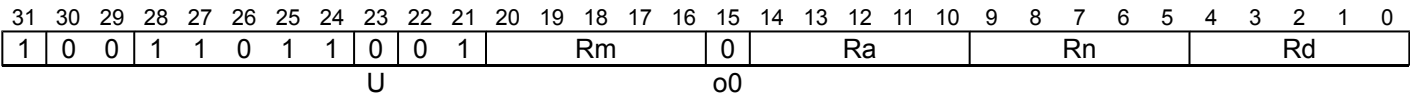
Operation

[SendEventLocal](#) ();

SMADDL

Signed Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias [SMULL](#).



SMADDL <Xd>, <Wn>, <Wm>, <Xa>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- <Xa> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
<a href="#">SMULL</a>	Ra == '11111'

Operation

```
bits(32) operand1 = X[n, 32];
bits(32) operand2 = X[m, 32];
bits(64) operand3 = X[a, 64];

integer result;

result = Int(operand3, FALSE) + (Int(operand1, FALSE) * Int(operand2, FALSE));

X[d, 64] = result<63:0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

SMAX (immediate)

Signed Maximum (immediate) determines the signed maximum of the source register value and immediate, and writes the result to the destination register.

Integer  
(FEAT\_CSSC)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	0	0	0	1	1	1	0	0	0	0	imm8								Rn				Rd					

32-bit (sf == 0)

```
SMAX <Wd>, <Wn>, #<sim>
```

64-bit (sf == 1)

```
SMAX <Xd>, <Xn>, #<sim>
```

```
if !HaveCSSC() then UNDEFINED;
integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm8);
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <sim> Is a signed immediate, in the range -128 to 127, encoded in the "imm8" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
integer result = Max(SInt(operand1), imm);
X[d, datasize] = result<datasize-1:0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

SMAX (register)

Signed Maximum (register) determines the signed maximum of the two source register values and writes the result to the destination register.

Integer  
(FEAT\_CSSC)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	1	1	0	0	0	Rn					Rd				

32-bit (sf == 0)

SMAX <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

SMAX <Xd>, <Xn>, <Xm>

```
if !HaveCSSC() then UNDEFINED;
integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
integer result = Max(SInt(operand1), SInt(operand2));
X[d, datasize] = result<datasize-1:0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# SMC

Secure Monitor Call causes an exception to EL3.  
SMC is available only for software executing at EL1 or higher. It is UNDEFINED in EL0.  
If the values of *HCR\_EL2*.TSC and *SCR\_EL3*.SMD are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception, recording it in *ESR\_ELx*, using the EC value 0x17, that is taken to EL3.  
If the value of *HCR\_EL2*.TSC is 1 and EL2 is enabled in the current Security state, execution of an SMC instruction at EL1 generates an exception that is taken to EL2, regardless of the value of *SCR\_EL3*.SMD.  
If the value of *HCR\_EL2*.TSC is 0 and the value of *SCR\_EL3*.SMD is 1, the SMC instruction is UNDEFINED.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	0	0	0	imm16																0	0	0	1	1

SMC #<imm>

// Empty.

## Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

## Operation

```
AArch64.CheckForSMCUndefOrTrap(imm16);  
AArch64.CallSecureMonitor(imm16);
```

SMIN (immediate)

Signed Minimum (immediate) determines the signed minimum of the source register value and immediate, and writes the result to the destination register.

Integer  
(FEAT\_CSSC)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	0	0	0	1	1	1	0	0	1	0	imm8								Rn				Rd					

32-bit (sf == 0)

```
SMIN <Wd>, <Wn>, #<sim>
```

64-bit (sf == 1)

```
SMIN <Xd>, <Xn>, #<sim>
```

```
if !HaveCSSC() then UNDEFINED;
integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm8);
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <sim> Is a signed immediate, in the range -128 to 127, encoded in the "imm8" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
integer result = Min(SInt(operand1), imm);
X[d, datasize] = result<datasize-1:0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

SMIN (register)

Signed Minimum (register) determines the signed minimum of the two source register values and writes the result to the destination register.

Integer  
(FEAT\_CSSC)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	1	1	0	1	0	Rn					Rd				

32-bit (sf == 0)

SMIN <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

SMIN <Xd>, <Xn>, <Xm>

```
if !HaveCSSC() then UNDEFINED;
integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
integer result = Min(SInt(operand1), SInt(operand2));
X[d, datasize] = result<datasize-1:0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMNEGL

Signed Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of [SMSUBL](#). This means:

- The encodings in this description are named to match the encodings of [SMSUBL](#).
  - The description of [SMSUBL](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.
- |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |    |   |   |   |   |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|----|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8 | 7 | 6 | 5 | 4  | 3 | 2 | 1 | 0 |
| 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | Rm |    |    |    |    | 1  | 1  | 1  | 1  | 1  | 1  | Rn |   |   |   |   | Rd |   |   |   |   |
| U  |    |    |    |    |    |    |    |    |    |    | o0 |    |    |    |    | Ra |    |    |    |    |    |    |   |   |   |   |    |   |   |   |   |

SMNEGL <Xd>, <Wn>, <Wm>

is equivalent to

[SMSUBL](#) <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of [SMSUBL](#) gives the operational pseudocode for this instruction.

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

# SMSTART

Enables access to Streaming SVE mode and SME architectural state.  
SMSTART enters Streaming SVE mode, and enables the SME ZA storage.  
SMSTART SM enters Streaming SVE mode, but does not enable the SME ZA storage.  
SMSTART ZA enables the SME ZA storage, but does not cause an entry to Streaming SVE mode.

This is an alias of [MSR \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [MSR \(immediate\)](#).
- The description of [MSR \(immediate\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

## System (FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	1	0	0	0	x	x	1	0	1	1	1	1	1	1	1
op1												CRm				op2															

SMSTART {<option>}

is equivalent to

MSR <pstatefield>, #1

and is always the preferred disassembly.

## Assembler Symbols

<option> Is an optional mode, encoded in “CRm<2:1>”:

CRm<2:1>	<option>
00	RESERVED
01	SM
10	ZA
11	[no specifier]

<pstatefield> Is a PSTATE field name. For the MSR instruction, this is encoded in “op1:op2:CRm”:

op1	op2	CRm	<pstatefield>	Architectural Feature
000	00x	xxxx	<a href="#">SEE PSTATE</a>	-
000	010	xxxx	<a href="#">SEE PSTATE</a>	-
000	011	xxxx	UAO	FEAT_UAO
000	100	xxxx	PAN	FEAT_PAN
000	101	xxxx	SPSel	-
000	11x	xxxx	RESERVED	-
001	000	000x	ALLINT	FEAT_NMI
001	000	001x	PM	FEAT_EBEP
001	000	01xx	RESERVED	-
001	000	1xxx	RESERVED	-
001	001	xxxx	RESERVED	-
001	01x	xxxx	RESERVED	-
001	1xx	xxxx	RESERVED	-
010	xxx	xxxx	RESERVED	-
011	000	xxxx	RESERVED	-
011	001	xxxx	SSBS	FEAT_SSBS
011	010	xxxx	DIT	FEAT_DIT
011	011	000x	RESERVED	-
011	011	001x	SVCRSM	FEAT_SME
011	011	010x	SVCRZA	FEAT_SME
011	011	011x	SVCRSMZA	FEAT_SME
011	011	1xxx	RESERVED	-
011	100	xxxx	TCO	FEAT_MTE
011	101	xxxx	RESERVED	-
011	110	xxxx	DAIFSet	-
011	111	xxxx	DAIFClr	-
1xx	xxx	xxxx	RESERVED	-

For the SMSTART and SMSTOP aliases, this is encoded in "CRm<2:1>", where 0b01 specifies SVCRSM, 0b10 specifies SVCRZA, and 0b11 specifies SVCRSMZA.

## Operation

The description of [MSR \(immediate\)](#) gives the operational pseudocode for this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# SMSTOP

Disables access to Streaming SVE mode and SME architectural state.  
SMSTOP exits Streaming SVE mode, and disables the SME ZA storage.  
SMSTOP SM exits Streaming SVE mode, but does not disable the SME ZA storage.  
SMSTOP ZA disables the SME ZA storage, but does not cause an exit from Streaming SVE mode.

This is an alias of [MSR \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [MSR \(immediate\)](#).
- The description of [MSR \(immediate\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

## System (FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	1	0	0	0	x	x	0	0	1	1	1	1	1	1	1
op1												CRm						op2													

SMSTOP {<option>}

is equivalent to

MSR <pstatefield>, #0

and is always the preferred disassembly.

## Assembler Symbols

<option> Is an optional mode, encoded in “CRm<2:1>”:

CRm<2:1>	<option>
00	RESERVED
01	SM
10	ZA
11	[no specifier]

<pstatefield> Is a PSTATE field name. For the MSR instruction, this is encoded in “op1:op2:CRm”:

op1	op2	CRm	<pstatefield>	Architectural Feature
000	00x	xxxx	<a href="#">SEE PSTATE</a>	-
000	010	xxxx	<a href="#">SEE PSTATE</a>	-
000	011	xxxx	UAO	FEAT_UAO
000	100	xxxx	PAN	FEAT_PAN
000	101	xxxx	SPSel	-
000	11x	xxxx	RESERVED	-
001	000	000x	ALLINT	FEAT_NMI
001	000	001x	PM	FEAT_EBEP
001	000	01xx	RESERVED	-
001	000	1xxx	RESERVED	-
001	001	xxxx	RESERVED	-
001	01x	xxxx	RESERVED	-
001	1xx	xxxx	RESERVED	-
010	xxx	xxxx	RESERVED	-
011	000	xxxx	RESERVED	-
011	001	xxxx	SSBS	FEAT_SSBS
011	010	xxxx	DIT	FEAT_DIT
011	011	000x	RESERVED	-
011	011	001x	SVCRSM	FEAT_SME
011	011	010x	SVCRZA	FEAT_SME
011	011	011x	SVCRSMZA	FEAT_SME
011	011	1xxx	RESERVED	-
011	100	xxxx	TCO	FEAT_MTE
011	101	xxxx	RESERVED	-
011	110	xxxx	DAIFSet	-
011	111	xxxx	DAIFClr	-
1xx	xxx	xxxx	RESERVED	-

For the SMSTART and SMSTOP aliases, this is encoded in "CRm<2:1>", where 0b01 specifies SVCRSM, 0b10 specifies SVCRZA, and 0b11 specifies SVCRSMZA.

## Operation

The description of [MSR \(immediate\)](#) gives the operational pseudocode for this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

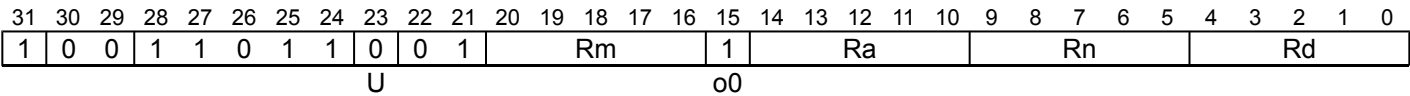
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SMSUBL

Signed Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias [SMNEGL](#).



SMSUBL <Xd>, <Wn>, <Wm>, <Xa>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- <Xa> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
<a href="#">SMNEGL</a>	Ra == '11111'

Operation

```
bits(32) operand1 = X[n, 32];
bits(32) operand2 = X[m, 32];
bits(64) operand3 = X[a, 64];

integer result;

result = Int(operand3, FALSE) - (Int(operand1, FALSE) * Int(operand2, FALSE));
X[d, 64] = result<63:0>;
```

Operational information

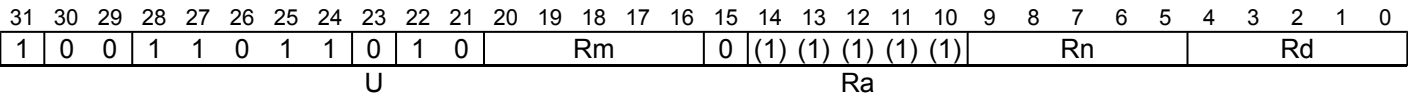
- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# SMULH

Signed Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.



SMULH <Xd>, <Xn>, <Xm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

## Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

## Operation

```
bits(64) operand1 = X[n, 64];
bits(64) operand2 = X[m, 64];

integer result;

result = Int(operand1, FALSE) * Int(operand2, FALSE);

X[d, 64] = result<127:64>;
```

## Operational information

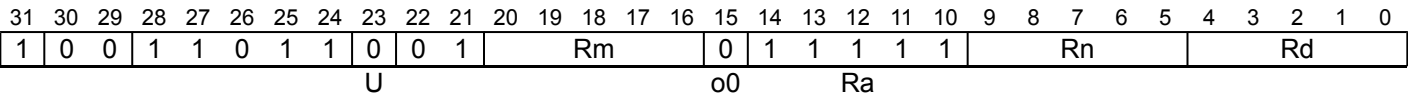
- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

# SMULL

Signed Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of [SMADDL](#). This means:

- The encodings in this description are named to match the encodings of [SMADDL](#).
- The description of [SMADDL](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



SMULL <Xd>, <Wn>, <Wm>

is equivalent to

SMADDL <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

## Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

## Operation

The description of [SMADDL](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSBB

Speculative Store Bypass Barrier is a memory barrier that prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions. For more information and details of the semantics, see *Speculative Store Bypass Barrier (SSBB)*.

This is an alias of [DSB](#). This means:

- The encodings in this description are named to match the encodings of [DSB](#).
- The description of [DSB](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	1	0	0	1	1	1	1	1
																CRm				opc											

SSBB

is equivalent to

[DSB](#) #0

and is always the preferred disassembly.

Operation

The description of [DSB](#) gives the operational pseudocode for this instruction.

## ST2G

Store Allocation Tags stores an Allocation Tag to two Tag granules of memory. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Signed offset](#)

### Post-index

(FEAT\_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	1	0	1	imm9									0	1	Xn				Xt					

**ST2G** <Xt|SP>, [<Xn|SP>], #<simm>

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

### Pre-index

(FEAT\_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	1	0	1	imm9									1	1	Xn				Xt					

**ST2G** <Xt|SP>, [<Xn|SP>, #<simm>]!

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

### Signed offset

(FEAT\_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	1	0	1	imm9									1	0	Xn				Xt					

**ST2G** <Xt|SP>, [<Xn|SP>{, #<simm>}]

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

### Assembler Symbols

- <Xt|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Xt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
- <simm> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

## Operation

```
bits(64) address;
bits(64) data = if t == 31 then SP[] else X[t, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(data);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_STORE);

AArch64.MemTag[address, accdesc] = tag;
AArch64.MemTag[address+TAG_GRANULE, accdesc] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST64B

Single-copy Atomic 64-byte Store without Return stores eight 64-bit doublewords from consecutive registers, Xt to X(t+7), to a memory location. The data that is stored is atomic and is required to be 64-byte-aligned.

Integer  
(FEAT\_LS64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	0	0	1	0	0	Rn				Rt					

ST64B <Xt>, [<Xn|SP> {, #0}]

```
if !HaveFeatLS64() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = MemOp_STORE;
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Xt> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckLDST64BEnabled();

bits(512) data;
bits(64) address;
bits(64) value;

AccessDescriptor accdesc = CreateAccDescLS64(memop, tagchecked);
for i = 0 to 7
    value = X[t+i, 64];
    if BigEndian(accdesc.acctype) then value = BigEndianReverse(value);
    data<63+64*i:64*i> = value;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

MemStore64B(address, data, accdesc);
```

ST64BV

Single-copy Atomic 64-byte Store with Return stores eight 64-bit doublewords from consecutive registers, Xt to X(t+7), to a memory location, and writes the status result of the store to a register. The data that is stored is atomic and is required to be 64-byte aligned.

Integer  
(FEAT\_LS64\_V)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	1	Rs				1	0	1	1	0	0	Rn				Rt						

ST64BV <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveFeatLS64_V() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = MemOp_STORE;
integer s = UInt(Rs);
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Xs>

Is the 64-bit name of the general-purpose register into which the status result of this instruction is written, encoded in the "Rs" field.  
The value returned is:  
**0xFFFFFFFF\_FFFFFFFF**  
If the memory location accessed does not support this instruction. In this case, the value at the memory location is UNKNOWN.  
  
**!= 0xFFFFFFFF\_FFFFFFFF**  
If the memory location accessed does support this instruction. In this case, the peripheral that provides the response defines the returned value and provides information on the state of the memory update at the memory location.  
  
If XZR is used, then the return value is ignored.
- <Xt>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.



## Operation

```
CheckST64BVEnabled();

bits(512) data;
bits(64) address;
bits(64) value;
bits(64) status;

AccessDescriptor accdesc = CreateAccDescLS64(memop, tagchecked);
for i = 0 to 7
    value = X[t+i, 64];
    if BigEndian(accdesc.acctype) then value = BigEndianReverse(value);
    data<63+64*i:64*i> = value;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

status = MemStore64BWithRet(address, data, accdesc);

if s != 31 then X[s, 64] = status;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST64BV0

Single-copy Atomic 64-byte EL0 Store with Return stores eight 64-bit doublewords from consecutive registers, Xt to X(t+7), to a memory location, with the bottom 32 bits taken from [ACCDATA\\_EL1](#), and writes the status result of the store to a register. The data that is stored is atomic and is required to be 64-byte aligned.

Integer  
(FEAT\_LS64\_ACCDATA)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	1	Rs				1	0	1	0	0	0	Rn				Rt						

ST64BV0 <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveFeatLS64_ACCDATA() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = MemOp_STORE;
integer s = UInt(Rs);
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Xs>

Is the 64-bit name of the general-purpose register into which the status result of this instruction is written, encoded in the "Rs" field.  
The value returned is:  
**0xFFFFFFFF\_FFFFFFFF**  
If the memory location accessed does not support this instruction. In this case, the value at the memory location is UNKNOWN.  
  
**!= 0xFFFFFFFF\_FFFFFFFF**  
If the memory location accessed does support this instruction. In this case, the peripheral that provides the response defines the returned value and provides information on the state of the memory update at the memory location.  
  
If XZR is used, then the return value is ignored.
- <Xt>

Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
CheckST64BV0Enabled();

bits(512) data;
bits(64) address;
bits(64) value;
bits(64) status;

AccessDescriptor accdesc = CreateAccDescLS64(memop, tagchecked);
bits(64) Xt = X[t, 64];
value<31:0> = ACCDATA_EL1<31:0>;
value<63:32> = Xt<63:32>;
if BigEndian(accdesc.acctype) then value = BigEndianReverse(value);
data<63:0> = value;
for i = 1 to 7
    value = X[t+i, 64];
    if BigEndian(accdesc.acctype) then value = BigEndianReverse(value);
    data<63+64*i:64*i> = value;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

status = MemStore64BWithRet(address, data, accdesc);

if s != 31 then X[s, 64] = status;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# STADD, STADDL

Atomic add on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADD does not have release semantics.
  - STADDL stores to memory with release semantics, as described in [Load-Acquire, Store-Release](#).
- For information about memory accesses, see [Load/Store addressing modes](#).

This is an alias of [LDADD, LDADDA, LDADDAL, LDADDL](#). This means:

- The encodings in this description are named to match the encodings of [LDADD, LDADDA, LDADDAL, LDADDL](#).
- The description of [LDADD, LDADDA, LDADDAL, LDADDL](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

## Integer (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs				0	0	0	0	0	0	Rn				1	1	1	1	1		
size				A				opc														Rt									

### 32-bit LDADD alias (size == 10 && R == 0)

STADD <Ws>, [<Xn|SP>]

is equivalent to

LDADD <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### 32-bit LDADDL alias (size == 10 && R == 1)

STADDL <Ws>, [<Xn|SP>]

is equivalent to

LDADDL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### 64-bit LDADD alias (size == 11 && R == 0)

STADD <Xs>, [<Xn|SP>]

is equivalent to

LDADD <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

### 64-bit LDADDL alias (size == 11 && R == 1)

STADDL <Xs>, [<Xn|SP>]

is equivalent to

LDADDL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

The description of [LDADD, LDADDA, LDADDAL, LDADDL](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STADDB, STADDLB

Atomic add on byte in memory, without return, atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDB does not have release semantics.
  - STADDLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.
- For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of [LDADDB, LDADDAB, LDADDALB, LDADDLB](#). This means:

- The encodings in this description are named to match the encodings of [LDADDB, LDADDAB, LDADDALB, LDADDLB](#).
- The description of [LDADDB, LDADDAB, LDADDALB, LDADDLB](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs				0	0	0	0	0	0	Rn				1	1	1	1	1		
size								A				opc								Rt											

No memory ordering (R == 0)

STADDB <Ws>, [[<Xn|SP>](#)]

is equivalent to

[LDADDB](#) <Ws>, WZR, [[<Xn|SP>](#)]

and is always the preferred disassembly.

Release (R == 1)

STADDLB <Ws>, [[<Xn|SP>](#)]

is equivalent to

[LDADDLB](#) <Ws>, WZR, [[<Xn|SP>](#)]

and is always the preferred disassembly.

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of [LDADDB, LDADDAB, LDADDALB, LDADDLB](#) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

STADDH, STADDLH

Atomic add on halfword in memory, without return, atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDH does not have release semantics.
  - STADDLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.
- For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDADDH, LDADDAH, LDADDALH, LDADDLH. This means:

- The encodings in this description are named to match the encodings of LDADDH, LDADDAH, LDADDALH, LDADDLH.
- The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs				0	0	0	0	0	0	Rn				1	1	1	1	1		
size								A				opc								Rt											

No memory ordering (R == 0)

STADDH <Ws>, [<Xn|SP>]

is equivalent to

LDADDH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STADDLH <Ws>, [<Xn|SP>]

is equivalent to

LDADDLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

# STCLR, STCLRL

Atomic bit clear on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLR does not have release semantics.
- STCLRL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of [LDCLR, LDCLRA, LDCLRAL, LDCLRL](#). This means:

- The encodings in this description are named to match the encodings of [LDCLR, LDCLRA, LDCLRAL, LDCLRL](#).
- The description of [LDCLR, LDCLRA, LDCLRAL, LDCLRL](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

## Integer (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	x	1	1	1	0	0	0	0	R	1			Rs			0	0	0	1	0	0					Rn		1	1	1	1	1
size								A				opc										Rt										

### 32-bit LDCLR alias (size == 10 && R == 0)

STCLR <Ws>, [<Xn|SP>]

is equivalent to

LDCLR <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### 32-bit LDCLRL alias (size == 10 && R == 1)

STCLRL <Ws>, [<Xn|SP>]

is equivalent to

LDCLRL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### 64-bit LDCLR alias (size == 11 && R == 0)

STCLR <Xs>, [<Xn|SP>]

is equivalent to

LDCLR <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

### 64-bit LDCLRL alias (size == 11 && R == 1)

STCLRL <Xs>, [<Xn|SP>]

is equivalent to

LDCLRL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.



## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

The description of [LDCLR](#), [LDCLRA](#), [LDCLRAL](#), [LDCLRL](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STCLRB, STCLRLB

Atomic bit clear on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLRB does not have release semantics.
  - STCLRLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.
- For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of [LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB](#). This means:

- The encodings in this description are named to match the encodings of [LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB](#).
- The description of [LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs				0	0	0	1	0	0	Rn				1	1	1	1	1	1	1
size								A				opc										Rt									

No memory ordering (R == 0)

STCLRB <Ws>, [<Xn|SP>]

is equivalent to

LDCLRB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STCLRLB <Ws>, [<Xn|SP>]

is equivalent to

LDCLRLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of [LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB](#) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

STCLR<sub>H</sub>, STCLR<sub>LH</sub>

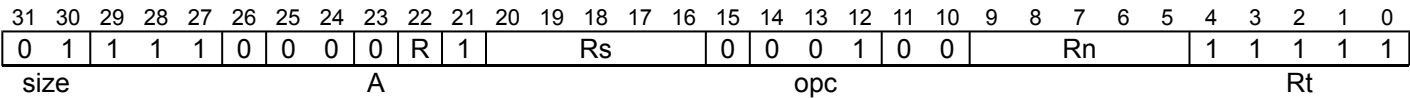
Atomic bit clear on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLR<sub>H</sub> does not have release semantics.
  - STCLR<sub>LH</sub> stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.
- For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of [LDCLR<sub>H</sub>](#), [LDCLR<sub>RAH</sub>](#), [LDCLR<sub>RALH</sub>](#), [LDCLR<sub>RLH</sub>](#). This means:

- The encodings in this description are named to match the encodings of [LDCLR<sub>H</sub>](#), [LDCLR<sub>RAH</sub>](#), [LDCLR<sub>RALH</sub>](#), [LDCLR<sub>RLH</sub>](#).
- The description of [LDCLR<sub>H</sub>](#), [LDCLR<sub>RAH</sub>](#), [LDCLR<sub>RALH</sub>](#), [LDCLR<sub>RLH</sub>](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer  
(FEAT\_LSE)



No memory ordering (R == 0)

STCLR<sub>H</sub> <Ws>, [[<Xn|SP>](#)]

is equivalent to

[LDCLR<sub>H</sub>](#) <Ws>, WZR, [[<Xn|SP>](#)]

and is always the preferred disassembly.

Release (R == 1)

STCLR<sub>LH</sub> <Ws>, [[<Xn|SP>](#)]

is equivalent to

[LDCLR<sub>LH</sub>](#) <Ws>, WZR, [[<Xn|SP>](#)]

and is always the preferred disassembly.

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of [LDCLR<sub>H</sub>](#), [LDCLR<sub>RAH</sub>](#), [LDCLR<sub>RALH</sub>](#), [LDCLR<sub>RLH</sub>](#) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

# STEOR, STEORL

Atomic Exclusive-OR on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory.

- STEOR does not have release semantics.
- STEORL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of [LDEOR, LDEORA, LDEORAL, LDEORL](#). This means:

- The encodings in this description are named to match the encodings of [LDEOR, LDEORA, LDEORAL, LDEORL](#).
- The description of [LDEOR, LDEORA, LDEORAL, LDEORL](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

## Integer (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs				0	0	1	0	0	0	Rn				1	1	1	1	1		
size								A				opc								Rt											

### 32-bit LDEOR alias (size == 10 && R == 0)

STEOR <Ws>, [<Xn|SP>]

is equivalent to

LDEOR <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### 32-bit LDEORL alias (size == 10 && R == 1)

STEORL <Ws>, [<Xn|SP>]

is equivalent to

LDEORL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### 64-bit LDEOR alias (size == 11 && R == 0)

STEOR <Xs>, [<Xn|SP>]

is equivalent to

LDEOR <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

### 64-bit LDEORL alias (size == 11 && R == 1)

STEORL <Xs>, [<Xn|SP>]

is equivalent to

LDEORL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

The description of [LDEOR](#), [LDEORA](#), [LDEORAL](#), [LDEORL](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# STEORB, STEORLB

Atomic Exclusive-OR on byte in memory, without return, atomically loads an 8-bit byte from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory.

- STEORB does not have release semantics.
  - STEORLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.
- For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of [LDEORB, LDEORAB, LDEORALB, LDEORLB](#). This means:

- The encodings in this description are named to match the encodings of [LDEORB, LDEORAB, LDEORALB, LDEORLB](#).
- The description of [LDEORB, LDEORAB, LDEORALB, LDEORLB](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

## Integer (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	1	1	0	0	0	0	R	1			Rs			0	0	1	0	0	0				Rn			1	1	1	1	1
size								A				opc										Rt										

### No memory ordering (R == 0)

STEORB <Ws>, [<Xn|SP>]

is equivalent to

LDEORB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### Release (R == 1)

STEORLB <Ws>, [<Xn|SP>]

is equivalent to

LDEORLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

## Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

The description of [LDEORB, LDEORAB, LDEORALB, LDEORLB](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

STEORH, STEORLH

Atomic Exclusive-OR on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory.

- STEORH does not have release semantics.
  - STEORLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.
- For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of [LDEORH, LDEORAH, LDEORALH, LDEORLH](#). This means:

- The encodings in this description are named to match the encodings of [LDEORH, LDEORAH, LDEORALH, LDEORLH](#).
- The description of [LDEORH, LDEORAH, LDEORALH, LDEORLH](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs					0	0	1	0	0	0	Rn					1	1	1	1	1
size								A				opc										Rt									

No memory ordering (R == 0)

STEORH <Ws>, [[<Xn|SP>](#)]

is equivalent to

[LDEORH](#) <Ws>, WZR, [[<Xn|SP>](#)]

and is always the preferred disassembly.

Release (R == 1)

STEORLH <Ws>, [[<Xn|SP>](#)]

is equivalent to

[LDEORLH](#) <Ws>, WZR, [[<Xn|SP>](#)]

and is always the preferred disassembly.

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of [LDEORH, LDEORAH, LDEORALH, LDEORLH](#) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

# STG

Store Allocation Tag stores an Allocation Tag to memory. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Signed offset](#)

## Post-index (FEAT\_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	0	0	1	imm9									0	1	Xn				Xt					

STG <Xt|SP>, [<Xn|SP>], #<simm>

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

## Pre-index (FEAT\_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	0	0	1	imm9									1	1	Xn				Xt					

STG <Xt|SP>, [<Xn|SP>, #<simm>]!

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

## Signed offset (FEAT\_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	0	0	1	imm9									1	0	Xn				Xt					

STG <Xt|SP>, [<Xn|SP>{, #<simm>}]

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

## Assembler Symbols

- <Xt|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Xt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
- <simm> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.



## Operation

```
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp\_STORE);

bits(64) data = if t == 31 then SP[] else X[t, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, accdesc] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STGM

Store Tag Multiple writes a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID\_EL1.BS, and the Allocation Tag written to address A is taken from the source register at  $4 * A < 7:4 > + 3:4 * A < 7:4 >$ .  
This instruction is UNDEFINED at EL0.  
This instruction generates an Unchecked access.

Integer  
(FEAT\_MTE2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	Xn				Xt					

STGM <Xt>, [<Xn|SP>]

```
if !HaveMTE2Ext() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);
```

Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Xt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

Operation

```
if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = X[t, 64];
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

integer size = 4 * (2 ^ (UInt(GMID_EL1.BS)));
address = Align(address, size);
integer count = size >> LOG2_TAG_GRANULE;
integer index = UInt(address < LOG2_TAG_GRANULE + 3: LOG2_TAG_GRANULE >);
AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_STORE);

for i = 0 to count-1
    bits(4) tag = data < (index*4)+3: index*4 >;
    AArch64.MemTag[address, accdesc] = tag;
    address = address + TAG_GRANULE;
    index = index + 1;
```

# STGP

Store Allocation Tag and Pair of registers stores an Allocation Tag and two 64-bit doublewords to memory, from two registers. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the base register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Signed offset](#)

## Post-index

(FEAT\_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	0	1	0	simm7							Xt2				Xn				Xt						

STGP <Xt1>, <Xt2>, [[<Xn|SP>](#)], #<imm>

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2\_TAG\_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

## Pre-index

(FEAT\_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	1	1	0	simm7							Xt2				Xn				Xt						

STGP <Xt1>, <Xt2>, [[<Xn|SP>](#), #<imm>]!

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2\_TAG\_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

## Signed offset

(FEAT\_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	1	0	0	simm7							Xt2				Xn				Xt						

STGP <Xt1>, <Xt2>, [[<Xn|SP>](#){, #<imm>}]

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2\_TAG\_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

## Assembler Symbols

<Xt1>            Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Xt" field.

- <Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Xt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
- <imm> For the post-index and pre-index variant: is the signed immediate offset, a multiple of 16 in the range -1024 to 1008, encoded in the "simm7" field.
- For the signed offset variant: is the optional signed immediate offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "simm7" field.

## Operation

```

bits(64) address;
bits(64) data1;
bits(64) data2;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data1 = X[t, 64];
data2 = X[t2, 64];

if !postindex then
    address = address + offset;

AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp\_STORE);

if !IsAligned(address, TAG\_GRANULE) then
    AArch64.Abort(address, AlignmentFault(accdesc));

Mem[address, 8, accdesc] = data1;
Mem[address+8, 8, accdesc] = data2;

AArch64.MemTag[address, accdesc] = AArch64.AllocationTagFromAddress(address);

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STILP

Store-Release ordered Pair of registers calculates an address from a base register value and an optional offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information on single-copy atomicity and alignment requirements, see *Requirements for single-copy atomicity* and *Alignment of data accesses*. The instruction also has memory ordering semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, with the additional requirement that:

- When using the pre-index addressing mode, the Memory effects associated with Xt2/Wt2 are Ordered-before the Memory effects associated with Xt1/Wt1.
- For all other addressing modes, the Memory effects associated with Xt1/Wt1 are Ordered-before the Memory effects associated with Xt2/Wt2.

For information about memory accesses, see *Load/Store addressing modes*.

Integer  
(FEAT\_LRCPC3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	1	1	0	0	1	0	0	0	Rt2				0	0	0	x	1	0	Rn				Rt						
size										L				opc2																	

32-bit (size == 10 && opc2 == 0001)

```
STILP <Wt1>, <Wt2>, [<Xn|SP>]
```

32-bit pre-index (size == 10 && opc2 == 0000)

```
STILP <Wt1>, <Wt2>, [<Xn|SP>, #-8]!
```

64-bit (size == 11 && opc2 == 0001)

```
STILP <Xt1>, <Xt2>, [<Xn|SP>]
```

64-bit pre-index (size == 11 && opc2 == 0000)

```
STILP <Xt1>, <Xt2>, [<Xn|SP>, #-16]!
```

```
boolean wback;
wback = opc2<0> == '0';
```

STILP has the same CONSTRAINED UNPREDICTABLE behavior as STP. For information about this CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STP*.

Assembler Symbols

- <Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Shared Decode

```
integer offset;
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer scale = 2 + UInt(size<0>);
integer datasize = 8 << scale;
offset = if opc2<0> == '0' then -1 * (2 << scale) else 0;

boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
  assert c IN {Constraint NONE, Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
  case c of
    when Constraint_NONE      rt_unknown = FALSE;      // value stored is pre-writeback
    when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
    when Constraint_UNDEF      UNDEFINED;
    when Constraint_NOP        EndOfInstruction();
```

## Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc = CreateAccDescAcgRel(MemOp_STORE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

if rt_unknown && t == n then
    data1 = bits(datasize) UNKNOWN;
else
    data1 = X[t, datasize];
if rt_unknown && t2 == n then
    data2 = bits(datasize) UNKNOWN;
else
    data2 = X[t2, datasize];

if HaveLSE2Ext() then
    bits(2*datasize) full_data;
    if BigEndian(accdesc.acctype) then
        full_data = data1:data2;
    else
        full_data = data2:data1;
    accdesc.ispair = TRUE;
    accdesc.highestaddressfirst = offset < 0;
    Mem[address, 2*dbytes, accdesc] = full_data;
else
    if offset < 0 then
        // Reverse the memory write order for negative pre-index.
        Mem[address+dbytes, dbytes, accdesc] = data2;
        Mem[address, dbytes, accdesc] = data1;
    else
        Mem[address, dbytes, accdesc] = data1;
        Mem[address+dbytes, dbytes, accdesc] = data2;
if wback then
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

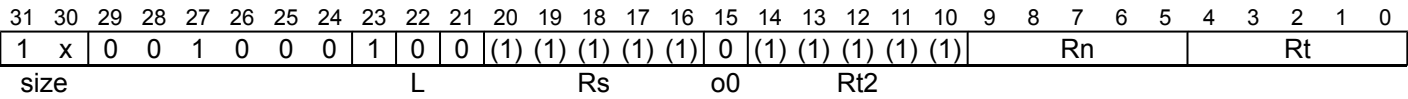
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLLR

Store LORelease Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

No offset  
(FEAT\_LOR)



32-bit (size == 10)

```
STLLR <Wt>, [<Xn|SP>{, #0}]
```

64-bit (size == 11)

```
STLLR <Xt>, [<Xn|SP>{, #0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

AccessDescriptor accdesc;
accdesc = CreateAccDescLOR(MemOp_STORE, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address;
data = X[t, elsize];
Mem[address, dbytes, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

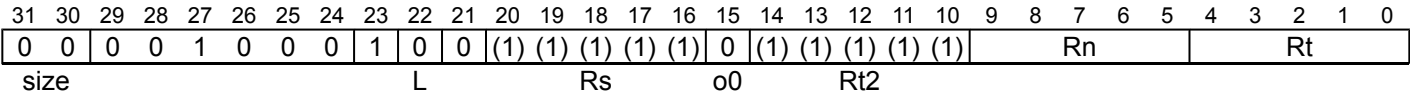
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



STLLRB

Store LORelease Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

No offset  
(FEAT\_LOR)



```
STLLRB <Wt>, [<Xn|SP>{, #0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescLOR(MemOp_STORE, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address;
data = X[t, 8];
Mem[address, 1, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

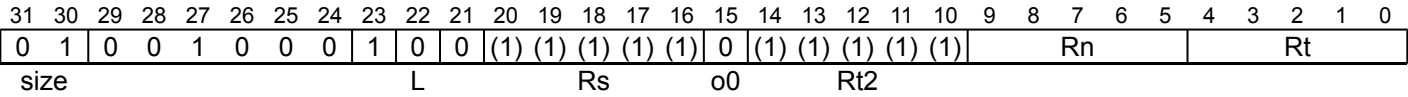
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLLRH

Store LORelease Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

No offset  
(FEAT\_LOR)



STLLRH <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescLOR(MemOp_STORE, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address;
data = X[t, 16];
Mem[address, 2, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLR

Store-Release Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 2 classes: [No offset](#) and [Pre-index](#)

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	1	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size		L							Rs					o0		Rt2															

32-bit (size == 10)

```
STLR <Wt>, [<Xn|SP>{, #0}]
```

64-bit (size == 11)

```
STLR <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
boolean wback = FALSE;
integer offset = 0;
boolean rt_unknown = FALSE;

integer elsize = 8 << UInt(size);
integer datasize = elsize;
boolean tagchecked = n != 31;
```

Pre-index
(FEAT\_LRCPC3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	Rn					Rt				
size																															

### 32-bit (size == 10)

```
STLR <Wt>, [<Xn|SP>, #-4]!
```

### 64-bit (size == 11)

```
STLR <Xt>, [<Xn|SP>, #-8]!
```

```
boolean wback = TRUE;

integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << UInt(size);
integer offset = -1 * (1 << UInt(size));
boolean tagchecked = TRUE;

boolean rt_unknown = FALSE;

if n == t && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
  assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
    when Constraint_UNKNOWN rt_unknown = TRUE;      // value stored is UNKNOWN
    when Constraint_UNDEF    UNDEFINED;
    when Constraint_NOP      EndOfInstruction();
```

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc;
accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

address = address + offset;
if rt_unknown then
  data = bits(datasize) UNKNOWN;
else
  data = X[t, datasize];
Mem[address, dbytes, accdesc] = data;

if wback then
  if n == 31 then
    SP[] = address;
  else
    X[n, 64] = address;
```

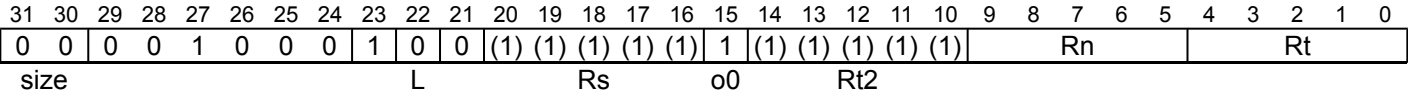
### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



# STLRB

Store-Release Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.



STLRB <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address;
data = X[t, 8];
Mem[address, 1, accdesc] = data;
```

## Operational information

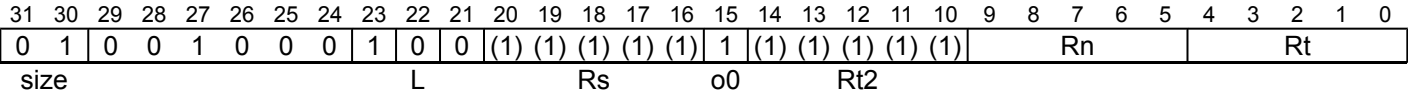
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# STLRH

Store-Release Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.



STLRH <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address;
data = X[t, 16];
Mem[address, 2, accdesc] = data;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

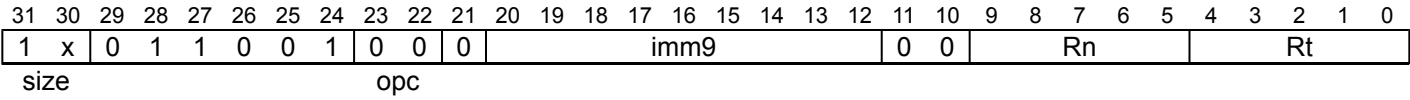
# STLUR

Store-Release Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register.

The instruction has memory ordering semantics as described in [Load-Acquire, Load-AcquirePC, and Store-Release](#)

For information about memory accesses, see [Load/Store addressing modes](#).

## Unscaled offset (FEAT\_LRCPC2)



### 32-bit (size == 10)

```
STLUR <Wt>, [<Xn|SP>{, #<sim>}]
```

### 64-bit (size == 11)

```
STLUR <Xt>, [<Xn|SP>{, #<sim>}]

integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

## Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;
boolean tagchecked = n != 31;
```

## Operation

```
bits(64) address;
bits(datasize) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, datasize];
Mem[address, datasize DIV 8, accdesc] = data;
```



## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

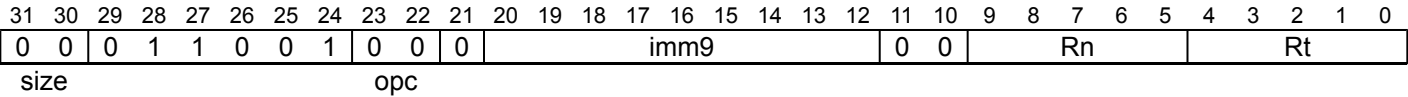
STLURB

Store-Release Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*

For information about memory accesses, see *Load/Store addressing modes*.

Unscaled offset  
(FEAT\_LRCPC2)



```
STLURB <Wt>, [<Xn|SP>{, #<simm>}]

bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 8];
Mem[address, 1, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

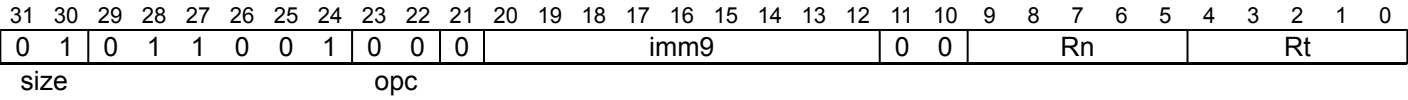
STLURH

Store-Release Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in [Load-Acquire, Load-AcquirePC, and Store-Release](#)

For information about memory accesses, see [Load/Store addressing modes](#).

Unscaled offset  
(FEAT\_LRCPC2)



```
STLURH <Wt>, [<Xn|SP>{, #<simm>}]

bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 16];
Mem[address, 2, accdesc] = data;
```

Operational information

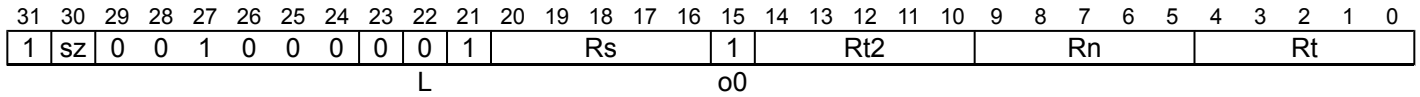
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STLXP

Store-Release Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords to a memory location if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). For information on single-copy atomicity and alignment requirements, see [Requirements for single-copy atomicity](#) and [Alignment of data accesses](#). If a 64-bit pair Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. The instruction also has memory ordering semantics, as described in [Load-Acquire, Store-Release](#). For information about memory accesses, see [Load/Store addressing modes](#).



### 32-bit (sz == 0)

```
STLXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

### 64-bit (sz == 1)

```
STLXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);      // ignored by all loads and store-release

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF    UNDEFINED;
        when Constraint_NOP      EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF    UNDEFINED;
        when Constraint_NOP      EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STLXP](#).

## Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
- 0 If the operation updates memory.
  - 1 If the operation fails to update memory.
- <Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

- <Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

#### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, TRUE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elseif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    bits(datasize DIV 2) el1 = X[t, datasize DIV 2];
    bits(datasize DIV 2) el2 = X[t2, datasize DIV 2];
    data = if BigEndian(accdesc.acctype) then el1:el2 else el2:el1;
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].

// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.
// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# STLXR

Store-Release Exclusive Register stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). The memory access is atomic. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	0	0	1	0	0	0	0	0	0	Rs				1	(1)	(1)	(1)	(1)	(1)	Rn				Rt						
size								L				o0				Rt2															

## 32-bit (size == 10)

STLXR <Ws>, <Wt>, [<Xn|SP>{, #0}]

## 64-bit (size == 11)

STLXR <Ws>, <Xt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

integer elsize = 8 << UInt(size);
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF    UNDEFINED;
        when Constraint_NOP      EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF    UNDEFINED;
        when Constraint_NOP      EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STLXR](#).

## Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.

- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

## Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, TRUE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elseif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t, elsize];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].

// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.
// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

## Operational information

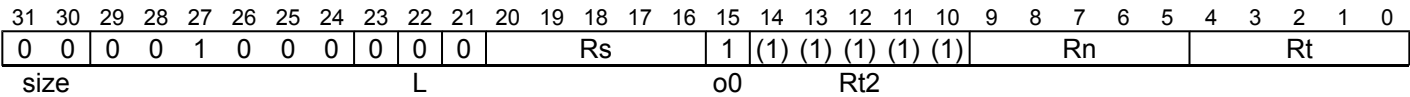
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# STLXRB

Store-Release Exclusive Register Byte stores a byte from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). The memory access is atomic. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses, see [Load/Store addressing modes](#).



STLXRB <Ws>, <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable DATAOVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint UNDEF    UNDEFINED;
        when Constraint NOP      EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable BASEOVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint UNDEF    UNDEFINED;
        when Constraint NOP      EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STLXRB](#).

## Assembler Symbols

- <Ws>
- Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
- 0
- If the operation updates memory.
- 1
- If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.



## Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, TRUE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elseif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t, 8];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].

// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.
// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, 1, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STLXRH

Store-Release Exclusive Register Halfword stores a halfword from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). The memory access is atomic. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#). For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	0	Rs				1	(1)	(1)	(1)	(1)	(1)	Rn				Rt						
size								L				o0				Rt2															

**STLXRH** <Ws>, <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable DATAOVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint UNDEF    UNDEFINED;
        when Constraint NOP      EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable BASEOVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint UNDEF    UNDEFINED;
        when Constraint NOP      EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STLXRH](#).

### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

**0**

If the operation updates memory.

**1**

If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

## Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, TRUE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t, 16];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].

// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.
// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, 2, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

## Operational information

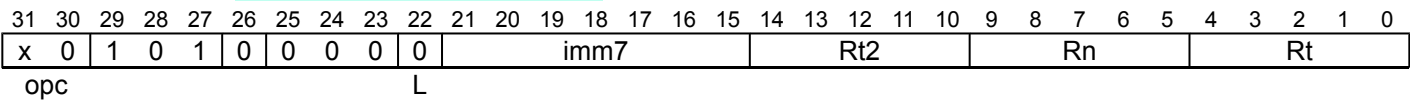
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# STNP

Store Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see [Load/Store addressing modes](#). For information about Non-temporal pair instructions, see [Load/Store Non-temporal pair](#).



## 32-bit (opc == 00)

```
STNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

## 64-bit (opc == 10)

```
STNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
// Empty.
```

## Assembler Symbols

- <Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.  
For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tagchecked = n != 31;
```

## Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean privileged = PSTATE.EL != ELO;

AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_STORE, TRUE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data1 = X[t, datasize];
data2 = X[t2, datasize];
Mem[address, dbytes, accdesc] = data1;
Mem[address+dbytes, dbytes, accdesc] = data2;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

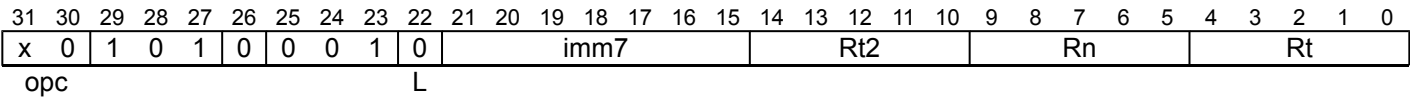
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STP

Store Pair of Registers calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Signed offset](#)

Post-index



32-bit (opc == 00)

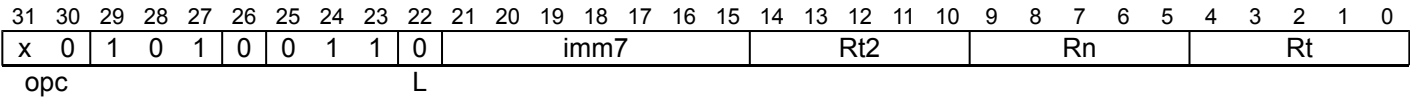
```
STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index



32-bit (opc == 00)

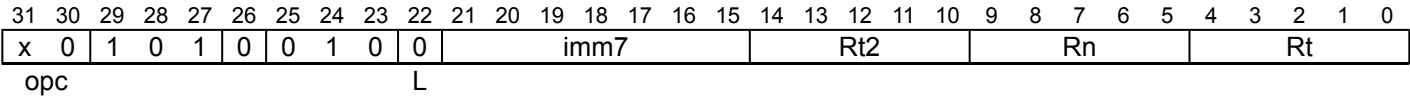
```
STP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset



32-bit (opc == 00)

```
STP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STP](#).

## Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.  For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.  For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.  For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE      rt_unknown = FALSE;      // value stored is pre-writeback
        when Constraint_UNKNOWN    rt_unknown = TRUE;       // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
```

## Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean privileged = PSTATE.EL != ELO;

AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

if rt_unknown && t == n then
    data1 = bits(datasize) UNKNOWN;
else
    data1 = X[t, datasize];
if rt_unknown && t2 == n then
    data2 = bits(datasize) UNKNOWN;
else
    data2 = X[t2, datasize];
if HaveLSE2Ext() then
    bits(2*datasize) full_data;
    if BigEndian(accdesc.acctype) then
        full_data = data1:data2;
    else
        full_data = data2:data1;
    accdesc.ispair = TRUE;
    Mem[address, 2*dbytes, accdesc] = full_data;
else
    Mem[address, dbytes, accdesc] = data1;
    Mem[address+dbytes, dbytes, accdesc] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## STR (immediate)

Store Register (immediate) stores a word or a doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	0	0	imm9									0	1	Rn				Rt					
size										opc																					

#### 32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>], #<sim>
```

#### 64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>], #<sim>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	x	1	1	1	0	0	0	0	0	0	imm9									1	1	Rn				Rt										
size										opc																										

#### 32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>, #<sim>]!
```

#### 64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>, #<sim>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	1	0	0	imm12												Rn				Rt					
size										opc																					

### 32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>{, #<pimm>}]
```

### 64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

## Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4. For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;
boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN  rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint_UNDEF    UNDEFINED;
        when Constraint_NOP      EndOfInstruction();
```

## Operation

```
bits(64) address;
bits(datasize) data;

boolean privileged = PSTATE.EL != ELO;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    data = X[t, datasize];
Mem[address, datasize DIV 8, accdesc] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

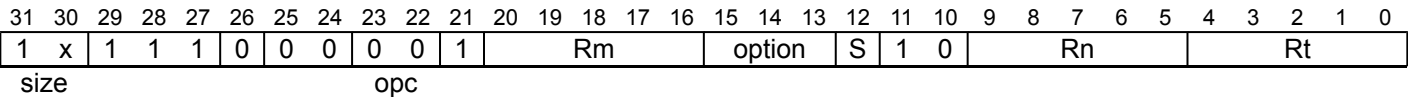
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see [Load/Store addressing modes](#).

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.



32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- <extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SCTX

- <amount> For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#2

- For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#3

## Shared Decode

```
integer n = UInt(Rn);  
integer t = UInt(Rt);  
integer m = UInt(Rm);  
  
integer datasize = 8 << scale;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);  
bits(64) address;  
bits(datasize) data;  
  
boolean privileged = PSTATE.EL != EL0;  
AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_STORE, FALSE, privileged, TRUE);  
  
if n == 31 then  
    CheckSPAlignment();  
    address = SP[];  
else  
    address = X[n, 64];  
  
address = address + offset;  
  
data = X[t, datasize];  
Mem[address, datasize DIV 8, accdesc] = data;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

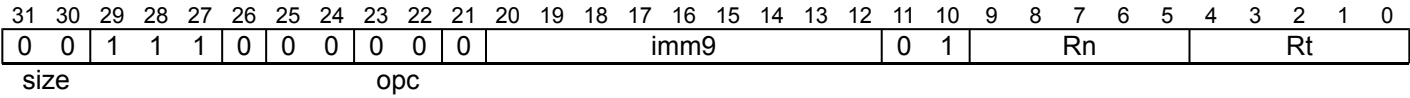
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STRB (immediate)

Store Register Byte (immediate) stores the least significant byte of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Unsigned offset](#)

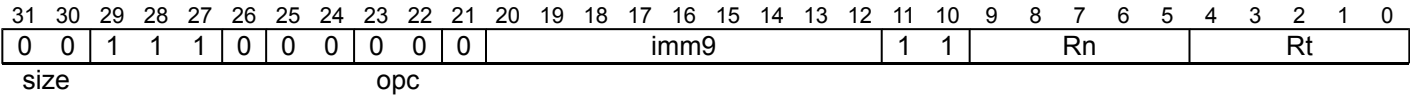
### Post-index



STRB <Wt>, [<Xn|SP>], #<simm>

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

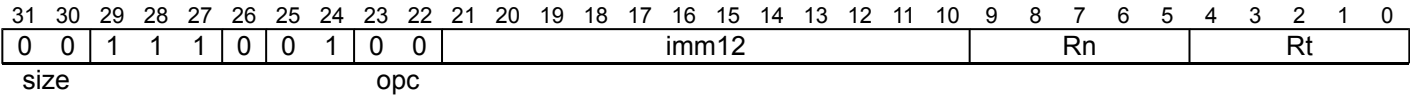
### Pre-index



STRB <Wt>, [<Xn|SP>, #<simm>]!

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset



STRB <Wt>, [<Xn|SP>{, #<pimm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STRB \(immediate\)](#).

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- <pimm> Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable\_WBOVERLAPST);
    assert c IN {Constraint\_NONE, Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP};
    case c of
        when Constraint\_NONE      rt_unknown = FALSE;    // value stored is original value
        when Constraint\_UNKNOWN    rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint\_UNDEF      UNDEFINED;
        when Constraint\_NOP        EndOfInstruction();
```

## Operation

```
bits(64) address;
bits(8) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t, 8];
Mem[address, 1, accdesc] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

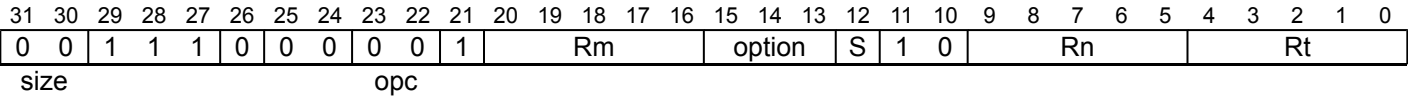
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a 32-bit register to the calculated address. For information about memory accesses, see [Load/Store addressing modes](#).

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.



Extended register (option != 011)

```
STRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Shifted register (option == 011)

```
STRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- <extend> Is the index extend specifier, encoded in "option":

option	<extend>
010	UXTW
110	SXTW
111	SCTX

<amount> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```



## Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0, 64);
bits(64) address;
bits(8) data;

boolean privileged = PSTATE.EL != ELO;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 8];
Mem[address, 1, accdesc] = data;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

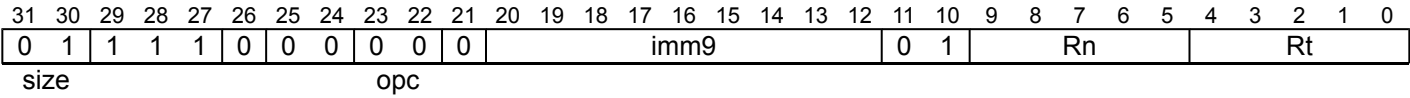
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STRH (immediate)

Store Register Halfword (immediate) stores the least significant halfword of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see [Load/Store addressing modes](#).

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Unsigned offset](#)

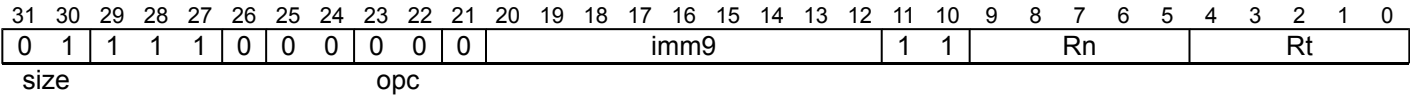
### Post-index



**STRH** <Wt>, [<Xn|SP>], #<simm>

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

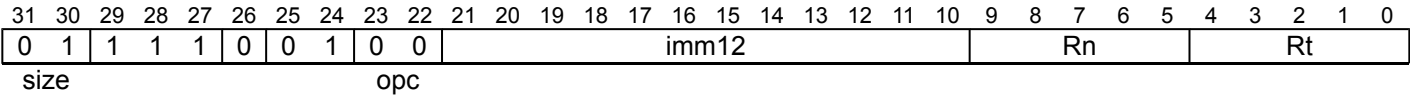
### Pre-index



**STRH** <Wt>, [<Xn|SP>, #<simm>]!

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset



**STRH** <Wt>, [<Xn|SP>{, #<pimm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STRH \(immediate\)](#).

### Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- <pimm> Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable\_WBOVERLAPST);
    assert c IN {Constraint\_NONE, Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP};
    case c of
        when Constraint\_NONE      rt_unknown = FALSE;    // value stored is original value
        when Constraint\_UNKNOWN    rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint\_UNDEF      UNDEFINED;
        when Constraint\_NOP        EndOfInstruction();
```

## Operation

```
bits(64) address;
bits(16) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t, 16];
Mem[address, 2, accdesc] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

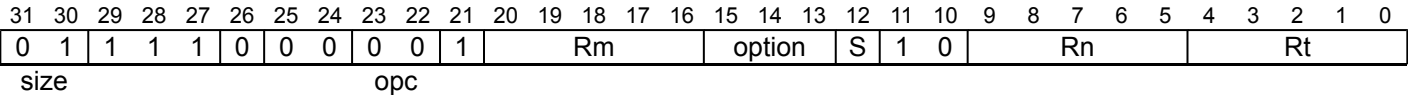
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a 32-bit register to the calculated address. For information about memory accesses, see [Load/Store addressing modes](#).

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.



```
STRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- <extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

- <amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#1

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(16) data;

boolean privileged = PSTATE.EL != ELO;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 16];
Mem[address, 2, accdesc] = data;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# STSET, STSETL

Atomic bit set on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSET does not have release semantics.
- STSETL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDSET, LDSETA, LDSETAL, LDSETL. This means:

- The encodings in this description are named to match the encodings of LDSET, LDSETA, LDSETAL, LDSETL.
- The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

## Integer (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs				0	0	1	1	0	0	Rn				1	1	1	1	1		
size				A								opc												Rt							

### 32-bit LDSET alias (size == 10 && R == 0)

STSET <Ws>, [<Xn|SP>]

is equivalent to

LDSET <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### 32-bit LDSETL alias (size == 10 && R == 1)

STSETL <Ws>, [<Xn|SP>]

is equivalent to

LDSETL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### 64-bit LDSET alias (size == 11 && R == 0)

STSET <Xs>, [<Xn|SP>]

is equivalent to

LDSET <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

### 64-bit LDSETL alias (size == 11 && R == 1)

STSETL <Xs>, [<Xn|SP>]

is equivalent to

LDSETL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

The description of [LDSET](#), [LDSETA](#), [LDSETAL](#), [LDSETL](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STSETB, STSETLB

Atomic bit set on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETB does not have release semantics.
  - STSETLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.
- For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDSETB, LDSETAB, LDSETALB, LDSETLB. This means:

- The encodings in this description are named to match the encodings of LDSETB, LDSETAB, LDSETALB, LDSETLB.
- The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs				0	0	1	1	0	0	Rn				1	1	1	1	1		
size								A				opc								Rt											

No memory ordering (R == 0)

STSETB <Ws>, [<Xn|SP>]

is equivalent to

LDSETB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STSETLB <Ws>, [<Xn|SP>]

is equivalent to

LDSETLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



STSETH, STSETLH

Atomic bit set on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETH does not have release semantics.
  - STSETLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.
- For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDSETH, LDSETAH, LDSETALH, LDSETLH. This means:

- The encodings in this description are named to match the encodings of LDSETH, LDSETAH, LDSETALH, LDSETLH.
- The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs					0	0	1	1	0	0	Rn					1	1	1	1	1
size								A				opc										Rt									

No memory ordering (R == 0)

STSETH <Ws>, [<Xn|SP>]

is equivalent to

LDSETH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STSETLH <Ws>, [<Xn|SP>]

is equivalent to

LDSETLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

## STSMAX, STSMAXL

Atomic signed maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAX does not have release semantics.
- STSMAXL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of [LDSMAX](#), [LDSMAXA](#), [LDSMAXAL](#), [LDSMAXL](#). This means:

- The encodings in this description are named to match the encodings of [LDSMAX](#), [LDSMAXA](#), [LDSMAXAL](#), [LDSMAXL](#).
- The description of [LDSMAX](#), [LDSMAXA](#), [LDSMAXAL](#), [LDSMAXL](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

### Integer (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs				0	1	0	0	0	0	Rn				1	1	1	1	1		
size								A				opc								Rt											

#### 32-bit LDSMAX alias (size == 10 && R == 0)

STSMAX <Ws>, [<Xn|SP>]

is equivalent to

LDSMAX <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 32-bit LDSMAXL alias (size == 10 && R == 1)

STSMAXL <Ws>, [<Xn|SP>]

is equivalent to

LDSMAXL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 64-bit LDSMAX alias (size == 11 && R == 0)

STSMAX <Xs>, [<Xn|SP>]

is equivalent to

LDSMAX <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 64-bit LDSMAXL alias (size == 11 && R == 1)

STSMAXL <Xs>, [<Xn|SP>]

is equivalent to

LDSMAXL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

The description of [LDSMAX](#), [LDSMAXA](#), [LDSMAXAL](#), [LDSMAXL](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STSMAXB, STSMAXB

Atomic signed maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAXB does not have release semantics.
- STSMAXB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB. This means:

- The encodings in this description are named to match the encodings of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB.
- The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs				0	1	0	0	0	0	Rn				1	1	1	1	1		
size								A				opc								Rt											

No memory ordering (R == 0)

STSMAXB <Ws>, [<Xn|SP>]

is equivalent to

LDSMAXB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STSMAXB <Ws>, [<Xn|SP>]

is equivalent to

LDSMAXLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

- <Ws>

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

STSMAXH, STSMAXLH

Atomic signed maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAXH does not have release semantics.
- STSMAXLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH. This means:

- The encodings in this description are named to match the encodings of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH.
- The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs				0	1	0	0	0	0	Rn				1	1	1	1	1		
size								A				opc								Rt											

No memory ordering (R == 0)

STSMAXH <Ws>, [<Xn|SP>]

is equivalent to

LDSMAXH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STSMAXLH <Ws>, [<Xn|SP>]

is equivalent to

LDSMAXLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

STSMIN, STSMINL

Atomic signed minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMIN does not have release semantics.
- STSMINL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDSMIN, LDSMINA, LDSMINAL, LDSMINL. This means:

- The encodings in this description are named to match the encodings of LDSMIN, LDSMINA, LDSMINAL, LDSMINL.
- The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs				0	1	0	1	0	0	Rn				1	1	1	1	1		
size								A				opc								Rt											

32-bit LDSMIN alias (size == 10 && R == 0)

STSMIN <Ws>, [<Xn|SP>]

is equivalent to

LDSMIN <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

32-bit LDSMINL alias (size == 10 && R == 1)

STSMINL <Ws>, [<Xn|SP>]

is equivalent to

LDSMINL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

64-bit LDSMIN alias (size == 11 && R == 0)

STSMIN <Xs>, [<Xn|SP>]

is equivalent to

LDSMIN <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

64-bit LDSMINL alias (size == 11 && R == 1)

STSMINL <Xs>, [<Xn|SP>]

is equivalent to

LDSMINL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

The description of [LDSDMIN](#), [LDSDMINA](#), [LDSDMINAL](#), [LDSDMINL](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STSMINB, STSMINLB

Atomic signed minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMINB does not have release semantics.
- STSMINLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB. This means:

- The encodings in this description are named to match the encodings of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB.
- The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs				0	1	0	1	0	0	Rn				1	1	1	1	1		
size								A				opc								Rt											

No memory ordering (R == 0)

STSMINB <Ws>, [<Xn|SP>]

is equivalent to

LDSMINB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STSMINLB <Ws>, [<Xn|SP>]

is equivalent to

LDSMINLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



STSMINH, STSMINLH

Atomic signed minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMINH does not have release semantics.
- STSMINLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH. This means:

- The encodings in this description are named to match the encodings of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH.
- The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs				0	1	0	1	0	0	Rn				1	1	1	1	1		
size								A				opc								Rt											

No memory ordering (R == 0)

STSMINH <Ws>, [<Xn|SP>]  
is equivalent to  
LDSMINH <Ws>, WZR, [<Xn|SP>]  
and is always the preferred disassembly.

Release (R == 1)

STSMINLH <Ws>, [<Xn|SP>]  
is equivalent to  
LDSMINLH <Ws>, WZR, [<Xn|SP>]  
and is always the preferred disassembly.

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

STTR

Store Register (unprivileged) stores a word or doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR\_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	x	1	1	1	0	0	0	0	0	0	imm9									1	0	Rn					Rt								
size										opc																									

32-bit (size == 10)

```
STTR <Wt>, [<Xn|SP>{, #<sim>}]
```

64-bit (size == 11)

```
STTR <Xt>, [<Xn|SP>{, #<sim>}]
```

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;
boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(datasize) data;

boolean privileged = AArch64.IsUnprivAccessPriv();
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, datasize];
Mem[address, datasize DIV 8, accdesc] = data;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

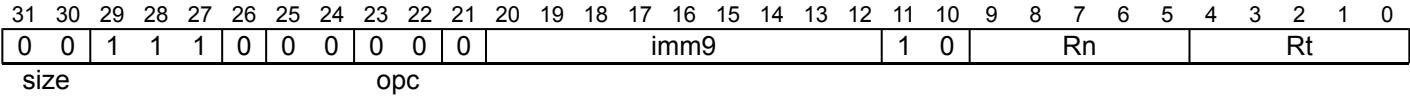
STTRB

Store Register Byte (unprivileged) stores a byte from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR\_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



STTRB <Wt>, [<Xn|SP>{, #<sim>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(8) data;

boolean privileged = AArch64.IsUnprivAccessPriv();
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 8];
Mem[address, 1, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STTRH

Store Register Halfword (unprivileged) stores a halfword from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR\_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	0	0	imm9										1	0	Rn					Rt			
size											opc																				

STTRH <Wt>, [<Xn|SP>{, #<sim>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt>
- Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>
- Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim>
- Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(16) data;

boolean privileged = AArch64.IsUnprivAccessPriv();
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 16];
Mem[address, 2, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

# STUMAX, STUMAXL

Atomic unsigned maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAX does not have release semantics.
- STUMAXL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of [LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL](#). This means:

- The encodings in this description are named to match the encodings of [LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL](#).
- The description of [LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

## Integer (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs				0	1	1	0	0	0	Rn				1	1	1	1	1		
size								A				opc								Rt											

### 32-bit LDUMAX alias (size == 10 && R == 0)

STUMAX <Ws>, [[Xn](#)|SP]

is equivalent to

[LDUMAX](#) <Ws>, WZR, [[Xn](#)|SP]

and is always the preferred disassembly.

### 32-bit LDUMAXL alias (size == 10 && R == 1)

STUMAXL <Ws>, [[Xn](#)|SP]

is equivalent to

[LDUMAXL](#) <Ws>, WZR, [[Xn](#)|SP]

and is always the preferred disassembly.

### 64-bit LDUMAX alias (size == 11 && R == 0)

STUMAX <Xs>, [[Xn](#)|SP]

is equivalent to

[LDUMAX](#) <Xs>, XZR, [[Xn](#)|SP]

and is always the preferred disassembly.

### 64-bit LDUMAXL alias (size == 11 && R == 1)

STUMAXL <Xs>, [[Xn](#)|SP]

is equivalent to

[LDUMAXL](#) <Xs>, XZR, [[Xn](#)|SP]

and is always the preferred disassembly.

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

The description of [LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STUMAXB, STUMAXLB

Atomic unsigned maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAXB does not have release semantics.
- STUMAXLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB. This means:

- The encodings in this description are named to match the encodings of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB.
- The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs				0	1	1	0	0	0	Rn				1	1	1	1	1		
size								A				opc								Rt											

No memory ordering (R == 0)

STUMAXB <Ws>, [<Xn|SP>]

is equivalent to

LDUMAXB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STUMAXLB <Ws>, [<Xn|SP>]

is equivalent to

LDUMAXLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



# STUMAXH, STUMAXLH

Atomic unsigned maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAXH does not have release semantics.
- STUMAXLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of [LDUMAXH](#), [LDUMAXAH](#), [LDUMAXALH](#), [LDUMAXLH](#). This means:

- The encodings in this description are named to match the encodings of [LDUMAXH](#), [LDUMAXAH](#), [LDUMAXALH](#), [LDUMAXLH](#).
- The description of [LDUMAXH](#), [LDUMAXAH](#), [LDUMAXALH](#), [LDUMAXLH](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

## Integer (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs				0	1	1	0	0	0	Rn				1	1	1	1	1		
size								A				opc								Rt											

### No memory ordering (R == 0)

STUMAXH <Ws>, [[<Xn|SP>](#)]

is equivalent to

[LDUMAXH](#) <Ws>, WZR, [[<Xn|SP>](#)]

and is always the preferred disassembly.

### Release (R == 1)

STUMAXLH <Ws>, [[<Xn|SP>](#)]

is equivalent to

[LDUMAXLH](#) <Ws>, WZR, [[<Xn|SP>](#)]

and is always the preferred disassembly.

## Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

The description of [LDUMAXH](#), [LDUMAXAH](#), [LDUMAXALH](#), [LDUMAXLH](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

# STUMIN, STUMINL

Atomic unsigned minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMIN does not have release semantics.
- STUMINL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDUMIN, LDUMINA, LDUMINAL, LDUMINL. This means:

- The encodings in this description are named to match the encodings of LDUMIN, LDUMINA, LDUMINAL, LDUMINL.
- The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

## Integer (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs				0	1	1	1	0	0	Rn				1	1	1	1	1		
size								A				opc								Rt											

### 32-bit LDUMIN alias (size == 10 && R == 0)

STUMIN <Ws>, [<Xn|SP>]

is equivalent to

LDUMIN <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### 32-bit LDUMINL alias (size == 10 && R == 1)

STUMINL <Ws>, [<Xn|SP>]

is equivalent to

LDUMINL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### 64-bit LDUMIN alias (size == 11 && R == 0)

STUMIN <Xs>, [<Xn|SP>]

is equivalent to

LDUMIN <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

### 64-bit LDUMINL alias (size == 11 && R == 1)

STUMINL <Xs>, [<Xn|SP>]

is equivalent to

LDUMINL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

## Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

The description of [LDUMIN](#), [LDUMINA](#), [LDUMINAL](#), [LDUMINL](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STUMINB, STUMINLB

Atomic unsigned minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMINB does not have release semantics.
- STUMINLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB. This means:

- The encodings in this description are named to match the encodings of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB.
- The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	R	1	Rs				0	1	1	1	0	0	Rn				1	1	1	1	1		
size								A				opc								Rt											

No memory ordering (R == 0)

STUMINB <Ws>, [<Xn|SP>]  
is equivalent to  
LDUMINB <Ws>, WZR, [<Xn|SP>]  
and is always the preferred disassembly.

Release (R == 1)

STUMINLB <Ws>, [<Xn|SP>]  
is equivalent to  
LDUMINLB <Ws>, WZR, [<Xn|SP>]  
and is always the preferred disassembly.

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

# STUMINH, STUMINLH

Atomic unsigned minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMINH does not have release semantics.
- STUMINLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH. This means:

- The encodings in this description are named to match the encodings of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH.
- The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

## Integer (FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs				0	1	1	1	0	0	Rn				1	1	1	1	1		
size								A				opc								Rt											

### No memory ordering (R == 0)

STUMINH <Ws>, [<Xn|SP>]

is equivalent to

LDUMINH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### Release (R == 1)

STUMINLH <Ws>, [<Xn|SP>]

is equivalent to

LDUMINLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

## Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

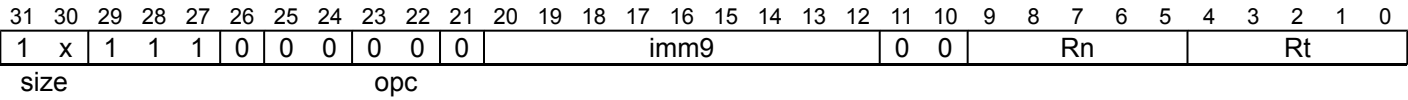
The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

STUR

Store Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see [Load/Store addressing modes](#).



32-bit (size == 10)

```
STUR <Wt>, [<Xn|SP>{, #<sim>}]
```

64-bit (size == 11)

```
STUR <Xt>, [<Xn|SP>{, #<sim>}]
```

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;
boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(datasize) data;

boolean privileged = PSTATE.EL != ELO;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, datasize];
Mem[address, datasize DIV 8, accdesc] = data;
```

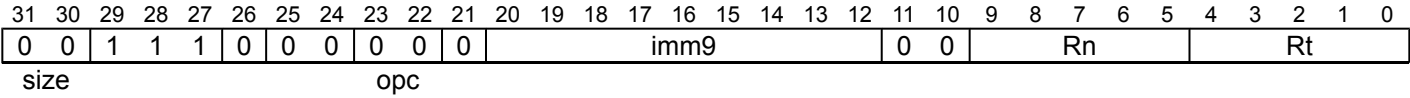
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



STURB

Store Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register. For information about memory accesses, see [Load/Store addressing modes](#).



STURB <Wt>, [<Xn|SP>{, #<simm>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(8) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 8];
Mem[address, 1, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

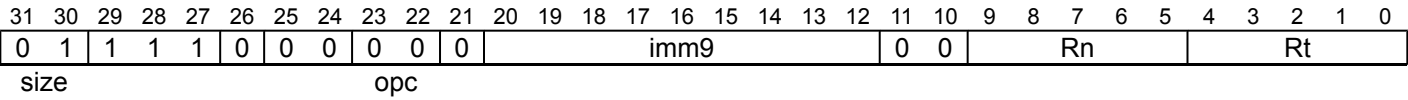
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



STURH

Store Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register. For information about memory accesses, see [Load/Store addressing modes](#).



STURH <Wt>, [<Xn|SP>{, #<simm>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(16) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 16];
Mem[address, 2, accdesc] = data;
```

Operational information

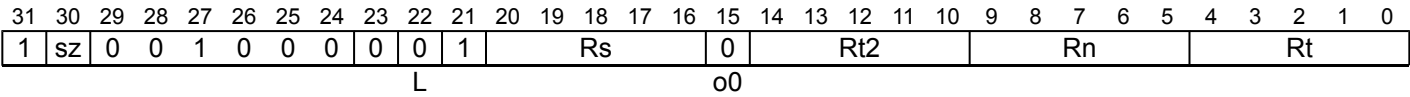
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# STXP

Store Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords from two registers to a memory location if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. For information on single-copy atomicity and alignment requirements, see *Requirements for single-copy atomicity* and *Alignment of data accesses*. If a 64-bit pair Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. For information about memory accesses, see *Load/Store addressing modes*.



## 32-bit (sz == 0)

STXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{, #0}]

## 64-bit (sz == 1)

STXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);      // ignored by all loads and store-release

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable DATAOVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint UNDEF    UNDEFINED;
        when Constraint NOP      EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable BASEOVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint UNDEF    UNDEFINED;
        when Constraint NOP      EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STXP*.

## Assembler Symbols

- <Ws>
- Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
- 0
- If the operation updates memory.
- 1
- If the operation fails to update memory.
- <Xt1>
- Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt2>
- Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Wt1>
- Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

#### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

## Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elseif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    bits(datasize DIV 2) el1 = X[t, datasize DIV 2];
    bits(datasize DIV 2) el2 = X[t2, datasize DIV 2];
    data = if BigEndian(accdesc.acctype) then el1:el2 else el2:el1;
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].

// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.
// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

## Operational information

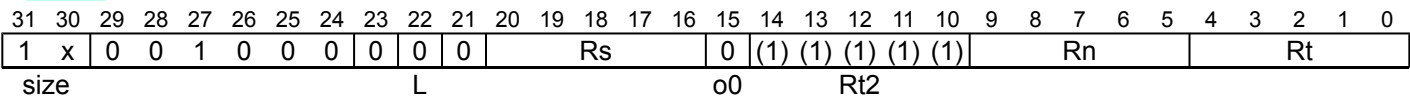
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# STXR

Store Exclusive Register stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). For information about memory accesses, see [Load/Store addressing modes](#).



## 32-bit (size == 10)

```
STXR <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

## 64-bit (size == 11)

```
STXR <Ws>, <Xt>, [<Xn|SP>{, #0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

integer elsize = 8 << UInt(size);
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF    UNDEFINED;
        when Constraint_NOP      EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF    UNDEFINED;
        when Constraint_NOP      EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STXR](#).

## Assembler Symbols

- <Ws>

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:  
**0**  
    If the operation updates memory.  
**1**  
    If the operation fails to update memory.
- <Xt>

Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Wt>

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is `IMPLEMENTATION DEFINED` whether the exception is generated.

If `AArch64.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is `IMPLEMENTATION DEFINED` whether the exception is generated.

## Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elseif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t, elsize];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].

// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.
// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

## Operational information

If `PSTATE.DIT` is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

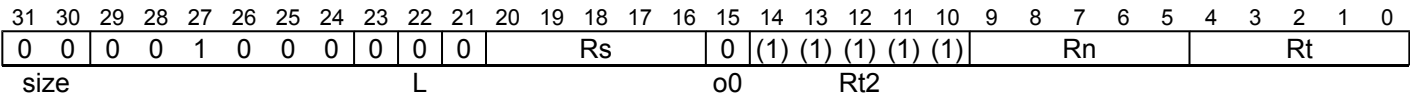
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# STXRB

Store Exclusive Register Byte stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). The memory access is atomic.

For information about memory accesses, see [Load/Store addressing modes](#).



STXRB <Ws>, <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable DATAOVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint UNDEF    UNDEFINED;
        when Constraint NOP      EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable BASEOVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint UNDEF    UNDEFINED;
        when Constraint NOP      EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [STXRB](#).

## Assembler Symbols

- <Ws>
- Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
- 0
- If the operation updates memory.
- 1
- If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

## Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elseif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t, 8];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].

// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.
// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, 1, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# STXRH

Store Exclusive Register Halfword stores a halfword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). The memory access is atomic.

For information about memory accesses, see [Load/Store addressing modes](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	0						Rs	0	(1)	(1)	(1)	(1)	(1)									Rt
size								L								o0								Rt2							

STXRH <Ws>, <Wt>, [<Xn|SP>{, #0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable DATAOVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint UNDEF    UNDEFINED;
        when Constraint NOP      EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable BASEOVERLAP);
    assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint UNDEF    UNDEFINED;
        when Constraint NOP      EndOfInstruction();
```

## Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:  

0 If the operation updates memory.  
1 If the operation fails to update memory.
- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.



## Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t, 16];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].

// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.
// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, 2, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STZ2G

Store Allocation Tags, Zeroing stores an Allocation Tag to two Tag granules of memory, zeroing the associated data locations. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Signed offset](#)

### Post-index

(FEAT\_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	1	1	1	imm9									0	1	Xn				Xt					

**STZ2G** <Xt|SP>, [<Xn|SP>], #<simm>

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

### Pre-index

(FEAT\_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	1	1	1	imm9									1	1	Xn				Xt					

**STZ2G** <Xt|SP>, [<Xn|SP>, #<simm>]!

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

### Signed offset

(FEAT\_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	1	1	1	imm9									1	0	Xn				Xt					

**STZ2G** <Xt|SP>, [<Xn|SP>{, #<simm>}]

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

### Assembler Symbols

- <Xt|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Xt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
- <simm> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

## Operation

```
bits(64) address;
bits(64) data = if t == 31 then SP[] else X[t, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(data);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_STORE);

if !IsAligned(address, TAG_GRANULE) then
    AArch64.Abort(address, AlignmentFault(accdesc));

Mem[address, TAG_GRANULE, accdesc] = Zeros(TAG_GRANULE * 8);
Mem[address+TAG_GRANULE, TAG_GRANULE, accdesc] = Zeros(TAG_GRANULE * 8);

AArch64.MemTag[address, accdesc] = tag;
AArch64.MemTag[address+TAG_GRANULE, accdesc] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# STZG

Store Allocation Tag, Zeroing stores an Allocation Tag to memory, zeroing the associated data location. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Signed offset](#)

## Post-index (FEAT\_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	0	1	1	imm9									0	1	Xn				Xt					

STZG <Xt|SP>, [<Xn|SP>], #<simm>

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

## Pre-index (FEAT\_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	0	1	1	imm9									1	1	Xn				Xt					

STZG <Xt|SP>, [<Xn|SP>, #<simm>]!

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

## Signed offset (FEAT\_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	0	1	1	imm9									1	0	Xn				Xt					

STZG <Xt|SP>, [<Xn|SP>{, #<simm>}]

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

## Assembler Symbols

- <Xt|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Xt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
- <simm> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

## Operation

```
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp\_STORE);

if !IsAligned(address, TAG\_GRANULE) then
    AArch64.Abort(address, AlignmentFault(accdesc));

Mem[address, TAG\_GRANULE, accdesc] = Zeros(TAG\_GRANULE * 8);

bits(64) data = if t == 31 then SP[] else X[t, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, accdesc] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STZGM

Store Tag and Zero Multiple writes a naturally aligned block of N Allocation Tags and stores zero to the associated data locations, where the size of N is identified in DCZID\_EL0.BS, and the Allocation Tag is taken from the source register bits<3:0>.

This instruction is UNDEFINED at EL0.  
This instruction generates an Unchecked access.

Integer  
(FEAT\_MTE2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	Xn				Xt					

STZGM <Xt>, [<Xn|SP>]

```
if !HaveMTE2Ext() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);
```

Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Xt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

Operation

```
if PSTATE.EL == EL0 then
    UNDEFINED;

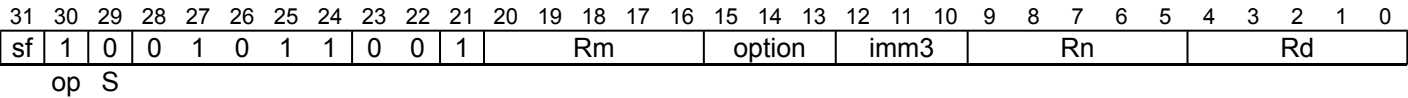
bits(64) data = X[t, 64];
bits(4) tag = data<3:0>;
bits(64) address;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

integer size = 4 * (2 ^ (UInt(DCZID_EL0.BS)));
address = Align(address, size);
integer count = size >> LOG2_TAG_GRANULE;
AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_STORE);

for i = 0 to count-1
    AArch64.MemTag[address, accdesc] = tag;
    Mem[address, TAG_GRANULE, accdesc] = Zeros(8 * TAG_GRANULE);
    address = address + TAG_GRANULE;
```

SUB (extended register)

Subtract (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.



32-bit (sf == 0)

```
SUB <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

64-bit (sf == 1)

```
SUB <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;
```

Assembler Symbols

- <Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in "option":

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	LSL UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);

operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');

if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## SUB (immediate)

Subtract (immediate) subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
sf		1	0	1	0	0	0	1	0	sh		imm12										Rn						Rd					
op S																																	

### 32-bit (sf == 0)

```
SUB <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

### 64-bit (sf == 1)

```
SUB <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case sh of
  when '0' imm = ZeroExtend(imm12, datasize);
  when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);
```

## Assembler Symbols

- <Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Wn|WSP> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <imm> Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- <shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #12

## Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(datasize) operand2;

operand2 = NOT(imm);
(result, -) = AddWithCarry(operand1, operand2, '1');

if d == 31 then
  SP[] = ZeroExtend(result, 64);
else
  X[d, datasize] = result;
```

## Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

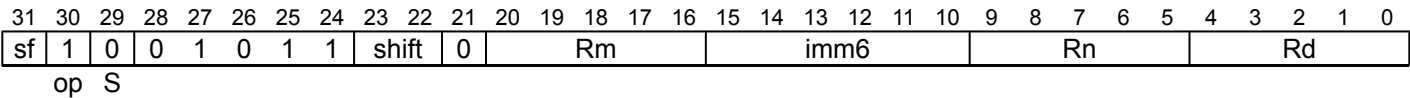
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUB (shifted register)

Subtract (shifted register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

This instruction is used by the alias [NEG \(shifted register\)](#).



32-bit (sf == 0)

```
SUB <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit (sf == 1)

```
SUB <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.  
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Alias Conditions

Alias	Is preferred when
<a href="#">NEG (shifted register)</a>	Rn == '11111'

## Operation

```
bits(datasize) result;  
bits(datasize) operand1 = X[n, datasize];  
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);  
  
operand2 = NOT(operand2);  
(result, -) = AddWithCarry(operand1, operand2, '1');  
  
X[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

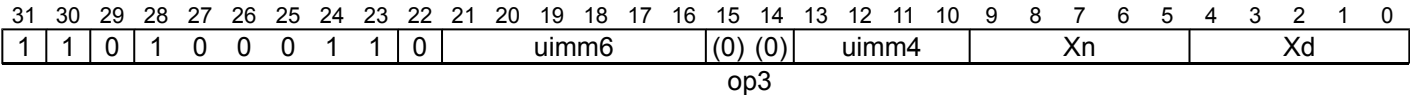
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUBG

Subtract with Tag subtracts an immediate value scaled by the Tag granule from the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR\_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

Integer  
(FEAT\_MTE)



SUBG <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>

```
if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
bits(64) offset = LSL(ZeroExtend(uimm6, 64), LOG2_TAG_GRANULE);
```

Assembler Symbols

- <Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.
- <Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Xn" field.
- <uimm6> Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the "uimm6" field.
- <uimm4> Is an unsigned immediate, in the range 0 to 15, encoded in the "uimm4" field.

Operation

```
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(4) start_tag = AArch64.AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;

if AArch64.AllocationTagAccessIsEnabled(PSTATE.EL) then
    rtag = AArch64.ChooseNonExcludedTag(start_tag, uimm4, exclude);
else
    rtag = '0000';

(result, -) = AddWithCarry(operand1, NOT(offset), '1');

result = AArch64.AddressWithAllocationTag(result, rtag);

if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;
```

SUBP

Subtract Pointer subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register.

Integer  
(FEAT\_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	0	1	1	0	Xm					0	0	0	0	0	0	Xn					Xd				

SUBP <Xd>, <Xn|SP>, <Xm|SP>

```
if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
- <Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
- <Xm|SP> Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

Operation

```
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) operand2 = if m == 31 then SP[] else X[m, 64];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = SignExtend(operand2<55:0>, 64);

bits(64) result;

operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');

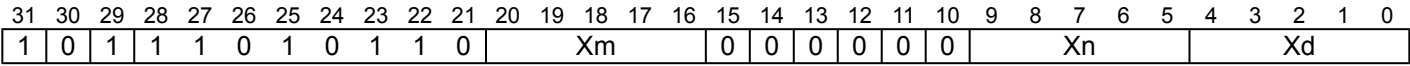
X[d, 64] = result;
```

SUBPS

Subtract Pointer, setting Flags subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register. It updates the condition flags based on the result of the subtraction.

This instruction is used by the alias [CMPP](#).

Integer  
(FEAT\_MTE)



SUBPS <Xd>, <Xn|SP>, <Xm|SP>

```
if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
- <Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
- <Xm|SP> Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

Alias Conditions

Alias	Is preferred when
<a href="#">CMPP</a>	S == '1' && Xd == '11111'

Operation

```
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) operand2 = if m == 31 then SP[] else X[m, 64];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = SignExtend(operand2<55:0>, 64);

bits(64) result;
bits(4) nzcvc;

operand2 = NOT(operand2);
(result, nzcvc) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcvc;
X[d, 64] = result;
```

SUBS (extended register)

Subtract (extended register), setting flags, subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias [CMP \(extended register\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	1	0	1	0	1	1	0	0	1	Rm					option			imm3			Rn					Rd				
op S																															

32-bit (sf == 0)

SUBS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}

64-bit (sf == 1)

SUBS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in "option":

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	LSL UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX



If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Alias Conditions

Alias	Is preferred when
<a href="#">CMP (extended register)</a>	Rd == '11111'

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);
bits(4) nzcvc;

operand2 = NOT(operand2);
(result, nzcvc) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcvc;

X[d, datasize] = result;
```

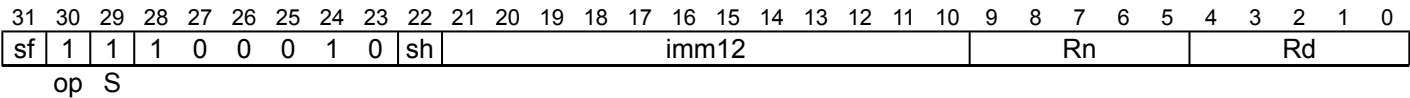
Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

SUBS (immediate)

Subtract (immediate), setting flags, subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias [CMP \(immediate\)](#).



32-bit (sf == 0)

```
SUBS <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit (sf == 1)

```
SUBS <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case sh of
  when '0' imm = ZeroExtend(imm12, datasize);
  when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);
```

Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn|WSP>

Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn|SP>

Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <imm>

Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- <shift>

Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #12

Alias Conditions

Alias	Is preferred when
<a href="#">CMP (immediate)</a>	Rd == '11111'

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[<datasize-1:0>] else X[n, datasize];
bits(datasize) operand2;
bits(4) nzcvc;

operand2 = NOT(imm);
(result, nzcvc) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcvc;

X[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

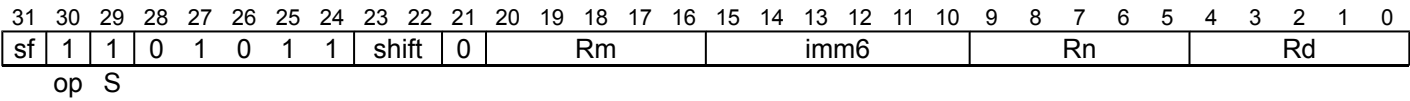
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUBS (shifted register)

Subtract (shifted register), setting flags, subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the aliases [CMP \(shifted register\)](#), and [NEGS](#).



32-bit (sf == 0)

```
SUBS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit (sf == 1)

```
SUBS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.  
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Alias Conditions

Alias	Is preferred when
<a href="#">CMP (shifted register)</a>	Rd == '11111'
<a href="#">NEGS</a>	Rn == '11111' && Rd != '11111'

## Operation

```
bits(datasize) result;  
bits(datasize) operand1 = X[n, datasize];  
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);  
bits(4) nzcvc;  
  
operand2 = NOT(operand2);  
(result, nzcvc) = AddWithCarry(operand1, operand2, '1');  
  
PSTATE.<N,Z,C,V> = nzcvc;  
  
X[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# SVC

Supervisor Call causes an exception to be taken to EL1.  
On executing an SVC instruction, the PE records the exception as a Supervisor Call exception in *ESR\_ELx*, using the EC value 0x15, and the value of the immediate argument.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	0	0	0	imm16																0	0	0	0	1

SVC #<imm>

// Empty.

## Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

## Operation

```
AArch64.CheckForSVCTrap(imm16);  
AArch64.CallSupervisor(imm16);
```

SWP, SWPA, SWPAL, SWPL

Swap word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, SWPA and SWPAL load from memory with acquire semantics.
- SWPL and SWPAL store to memory with release semantics.
- SWP has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about memory accesses, see Load/Store addressing modes.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	x	1	1	1	0	0	0	A	R	1	Rs				1	0	0	0	0	0	Rn				Rt						
size																															

**32-bit SWP (size == 10 && A == 0 && R == 0)**

SWP <Ws>, <Wt>, [<Xn|SP>]

**32-bit SWPA (size == 10 && A == 1 && R == 0)**

SWPA <Ws>, <Wt>, [<Xn|SP>]

**32-bit SWPAL (size == 10 && A == 1 && R == 1)**

SWPAL <Ws>, <Wt>, [<Xn|SP>]

**32-bit SWPL (size == 10 && A == 0 && R == 1)**

SWPL <Ws>, <Wt>, [<Xn|SP>]

**64-bit SWP (size == 11 && A == 0 && R == 0)**

SWP <Xs>, <Xt>, [<Xn|SP>]

**64-bit SWPA (size == 11 && A == 1 && R == 0)**

SWPA <Xs>, <Xt>, [<Xn|SP>]

**64-bit SWPAL (size == 11 && A == 1 && R == 1)**

SWPAL <Xs>, <Xt>, [<Xn|SP>]

**64-bit SWPL (size == 11 && A == 0 && R == 1)**

SWPL <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

**Assembler Symbols**

<Ws>	Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.



## Operation

```
bits(64) address;
bits(datasize) data;
bits(datasize) store_value;
AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

store_value = X[s, datasize];

bits(datasize) comparevalue = bits(datasize) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

X[t, regsize] = ZeroExtend(data, regsize);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SWPB, SWPAB, SWPALB, SWPLB

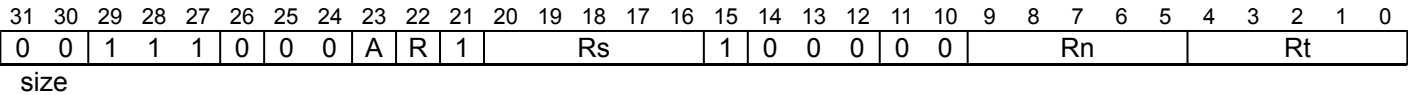
Swap byte in memory atomically loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAB and SWPALB load from memory with acquire semantics.
- SWPLB and SWPALB store to memory with release semantics.
- SWPB has neither acquire nor release semantics.

For more information about memory ordering semantics, see [Load-Acquire, Store-Release](#).

For information about memory accesses, see [Load/Store addressing modes](#).

Integer  
(FEAT\_LSE)



SWPAB (A == 1 && R == 0)

```
SWPAB <Ws>, <Wt>, [<Xn|SP>]
```

SWPALB (A == 1 && R == 1)

```
SWPALB <Ws>, <Wt>, [<Xn|SP>]
```

SWPB (A == 0 && R == 0)

```
SWPB <Ws>, <Wt>, [<Xn|SP>]
```

SWPLB (A == 0 && R == 1)

```
SWPLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(8) data;
bits(8) store_value;
AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

store_value = X[s, 8];

bits(8) comparevalue = bits(8) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

X[t, 32] = ZeroExtend(data, 32);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SWPH, SWPAH, SWPALH, SWPLH

Swap halfword in memory atomically loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAH and SWPALH load from memory with acquire semantics.
- SWPLH and SWPALH store to memory with release semantics.
- SWPH has neither acquire nor release semantics.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

Integer  
(FEAT\_LSE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	A	R	1	Rs				1	0	0	0	0	0	Rn				Rt						
size																															

SWPAH (A == 1 && R == 0)

SWPAH <Ws>, <Wt>, [<Xn|SP>]

SWPALH (A == 1 && R == 1)

SWPALH <Ws>, <Wt>, [<Xn|SP>]

SWPH (A == 0 && R == 0)

SWPH <Ws>, <Wt>, [<Xn|SP>]

SWPLH (A == 0 && R == 1)

SWPLH <Ws>, <Wt>, [<Xn|SP>]

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(16) data;
bits(16) store_value;
AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

store_value = X[s, 16];

bits(16) comparevalue = bits(16) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

X[t, 32] = ZeroExtend(data, 32);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

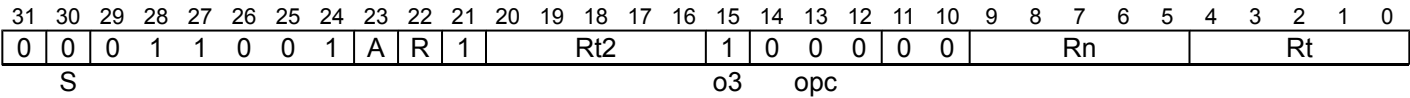
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# SWPP, SWPPA, SWPPAL, SWPPL

Swap quadword in memory atomically loads a 128-bit quadword from a memory location, and stores the value held in a pair of registers back to the same memory location. The value initially loaded from memory is returned in the same pair of registers.

- SWPPA and SWPPAL load from memory with acquire semantics.
- SWPPL and SWPPAL store to memory with release semantics.
- SWPP has neither acquire nor release semantics.

## Integer (FEAT\_LSE128)



### SWPP (A == 0 && R == 0)

SWPP <Xt1>, <Xt2>, [<Xn|SP>]

### SWPPA (A == 1 && R == 0)

SWPPA <Xt1>, <Xt2>, [<Xn|SP>]

### SWPPAL (A == 1 && R == 1)

SWPPAL <Xt1>, <Xt2>, [<Xn|SP>]

### SWPPL (A == 0 && R == 1)

SWPPL <Xt1>, <Xt2>, [<Xn|SP>]

```
if !HaveLSE128() then UNDEFINED;
if Rt == '11111' then UNDEFINED;
if Rt2 == '11111' then UNDEFINED;
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer n = UInt(Rn);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable LSE128OVERLAP);
  assert c IN {Constraint UNKNOWN, Constraint UNDEF, Constraint NOP};
  case c of
    when Constraint UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
    when Constraint UNDEF   UNDEFINED;
    when Constraint NOP     EndOfInstruction();
```

## Assembler Symbols

- <Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
bits(64) address;
bits(64) value1 = X[t, 64];
bits(64) value2 = X[t2, 64];
bits(128) data;
bits(128) store_value;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

store_value = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

bits(128) comparevalue = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

if rt_unknown then
    data = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = data<127:64>;
    X[t2, 64] = data<63:0>;
else
    X[t, 64] = data<63:0>;
    X[t2, 64] = data<127:64>;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of [SBFM](#). This means:

- The encodings in this description are named to match the encodings of [SBFM](#).
  - The description of [SBFM](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.
- |     |    |    |    |    |    |    |    |    |      |    |    |    |    |    |    |      |    |    |    |    |    |    |   |   |   |   |   |    |   |   |   |  |  |
|-----|----|----|----|----|----|----|----|----|------|----|----|----|----|----|----|------|----|----|----|----|----|----|---|---|---|---|---|----|---|---|---|--|--|
| 31  | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22   | 21 | 20 | 19 | 18 | 17 | 16 | 15   | 14 | 13 | 12 | 11 | 10 | 9  | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |  |  |
| sf  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | N    | 0  | 0  | 0  | 0  | 0  | 0  | 0    | 0  | 0  | 1  | 1  | 1  | Rn |   |   |   |   |   | Rd |   |   |   |  |  |
| opc |    |    |    |    |    |    |    |    | immr |    |    |    |    |    |    | imms |    |    |    |    |    |    |   |   |   |   |   |    |   |   |   |  |  |

32-bit (sf == 0 && N == 0)

SXTB <Wd>, <Wn>

is equivalent to

SBFM <Wd>, <Wn>, #0, #7

and is always the preferred disassembly.

64-bit (sf == 1 && N == 1)

SXTB <Xd>, <Wn>

is equivalent to

SBFM <Xd>, <Xn>, #0, #7

and is always the preferred disassembly.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of [SBFM](#) gives the operational pseudocode for this instruction.

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



SXTH

Sign Extend Halfword extracts a 16-bit value, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of [SBFM](#). This means:

- The encodings in this description are named to match the encodings of [SBFM](#).
- The description of [SBFM](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
sf	0	0	1	0	0	1	1	0	N	0	0	0	0	0	0	0	0	1	1	1	1	Rn						Rd					
opc									immr								imms																

32-bit (sf == 0 && N == 0)

SXTH <Wd>, <Wn>

is equivalent to

[SBFM](#) <Wd>, <Wn>, #0, #15

and is always the preferred disassembly.

64-bit (sf == 1 && N == 1)

SXTH <Xd>, <Wn>

is equivalent to

[SBFM](#) <Xd>, <Xn>, #0, #15

and is always the preferred disassembly.

Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xd>

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn>

Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Wn>

Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of [SBFM](#) gives the operational pseudocode for this instruction.

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

SXTW

Sign Extend Word sign-extends a word to the size of the register, and writes the result to the destination register.

This is an alias of [SBFM](#). This means:

- The encodings in this description are named to match the encodings of [SBFM](#).
- The description of [SBFM](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	0	0	1	0	0	1	1	0	1	0	0	0	0	0	0	0	1	1	1	1	1	Rn						Rd					
sf			opc								N		immr				imms																

64-bit

SXTW <Xd>, <Wn>

is equivalent to

SBFM <Xd>, <Xn>, #0, #31

and is always the preferred disassembly.

Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of [SBFM](#) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

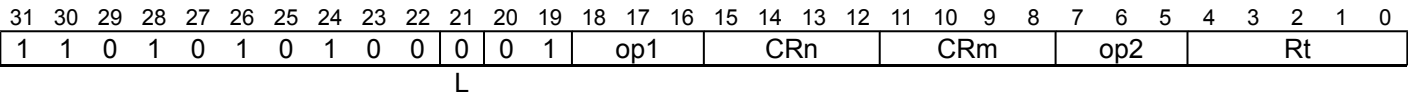
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SYS

System instruction. For more information, see [Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions](#) for the encodings of System instructions.

This instruction is used by the aliases [AT](#), [BRB](#), [CFP](#), [COSP](#), [CPP](#), [DC](#), [DVP](#), [GCSPOPCX](#), [GCSPOPX](#), [GCSPUSHM](#), [GCSPUSHX](#), [GCSSS1](#), [IC](#), [TLBI](#), and [TRCIT](#).



SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}

```
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler Symbols

- <op1>
- Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- <Cn>
- Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
- <Cm>
- Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- <op2>
- Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
- <Xt>
- Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Alias Conditions

Alias	Is preferred when
<a href="#">AT</a>	CRn == '0111' && CRm == '100x' && <a href="#">SysOp</a> (op1, '0111', CRm, op2) == <a href="#">Sys_AT</a>
<a href="#">BRB</a>	op1 == '001' && CRn == '0111' && CRm == '0010' && <a href="#">SysOp</a> ('001', '0111', '0010', op2) == <a href="#">Sys_</a>
<a href="#">CFP</a>	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '100'
<a href="#">COSP</a>	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '110'
<a href="#">CPP</a>	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '111'
<a href="#">DC</a>	CRn == '0111' && <a href="#">SysOp</a> (op1, '0111', CRm, op2) == <a href="#">Sys_DC</a>
<a href="#">DVP</a>	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '101'
<a href="#">GCSPOPCX</a>	op1 == '000' && CRn == '0111' && CRm == '0111' && op2 == '101'
<a href="#">GCSPOPX</a>	op1 == '000' && CRn == '0111' && CRm == '0111' && op2 == '110'
<a href="#">GCSPUSHM</a>	op1 == '011' && CRn == '0111' && CRm == '0111' && op2 == '000'
<a href="#">GCSPUSHX</a>	op1 == '000' && CRn == '0111' && CRm == '0111' && op2 == '100'
<a href="#">GCSSS1</a>	op1 == '011' && CRn == '0111' && CRm == '0111' && op2 == '010'
<a href="#">IC</a>	CRn == '0111' && <a href="#">SysOp</a> (op1, '0111', CRm, op2) == <a href="#">Sys_IC</a>
<a href="#">TLBI</a>	CRn == '100x' && <a href="#">SysOp</a> (op1, CRn, CRm, op2) == <a href="#">Sys_TLBI</a>
<a href="#">TRCIT</a>	op1 == '011' && CRn == '0111' && CRm == '0010' && op2 == '111'

Operation

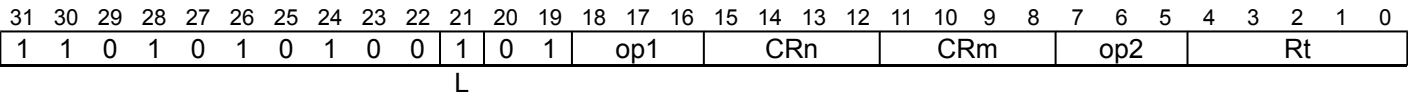
```
AArch64.SysInstr(1, sys_op1, sys_crn, sys_crm, sys_op2, t);
```



**SYSL**

System instruction with result. For more information, see *Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions* for the encodings of System instructions.

This instruction is used by the aliases [GCSPOPM](#), and [GCSSS2](#).



**SYSL** <Xt>, #<op1>, <Cn>, <Cm>, #<op2>

```
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

**Assembler Symbols**

- <Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.
- <op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- <Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
- <Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- <op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

**Alias Conditions**

Alias	Is preferred when
<a href="#">GCSPOPM</a>	op1 == '011' && CRn == '0111' && CRm == '0111' && op2 == '001'
<a href="#">GCSSS2</a>	op1 == '011' && CRn == '0111' && CRm == '0111' && op2 == '011'

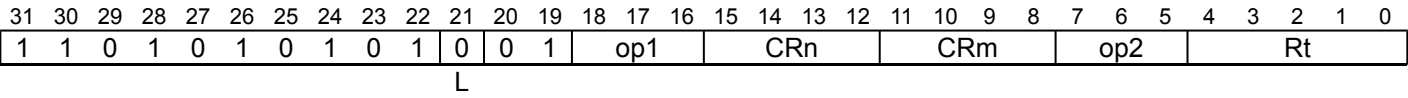
**Operation**

```
// No architecturally defined instructions here.
AArch64.SysInstrWithResult(1, sys_op1, sys_crn, sys_crm, sys_op2, t);
```

SYSP

128-bit System instruction.  
This instruction is used by the alias [TLBIP](#).

System  
(FEAT\_SYSPINSTR128)



SYSP #<op1>, <Cn>, <Cm>, #<op2>{, <Xt1>, <Xt2>}

```
if !HaveSysInstr128() then UNDEFINED;
if Rt<0> == '1' && Rt != '11111' then UNDEFINED;
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);
integer t2 = if t == 31 then 31 else UInt(Rt) + 1;

integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler Symbols

- <op1> Is a 3-bit unsigned immediate, in the range 0 to 6, encoded in the "op1" field.
- <Cn> Is a name 'Cn', with 'n' in the range 8 to 9, encoded in the "CRn" field.
- <Cm> Is a name 'Cm', with 'm' in the range 0 to 7, encoded in the "CRm" field.
- <op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
- <Xt1> Is the 64-bit name of the first optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.
- <Xt2> Is the 64-bit name of the second optional general-purpose source register, defaulting to '11111', encoded as "Rt" + 1. Defaults to '11111' if "Rt" = '11111'.

Alias Conditions

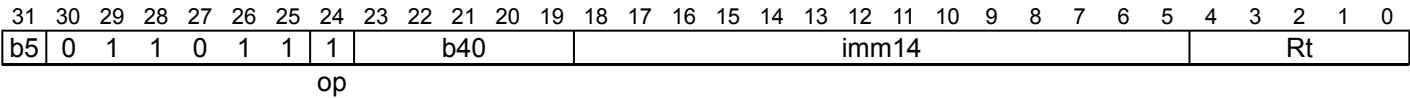
Alias	Is preferred when
<a href="#">TLBIP</a>	CRn == '100x' && SysOp(op1, CRn, CRm, op2) == <a href="#">Sys_TLBIP</a>

Operation

```
AArch64.SysInstr128(1, sys_op1, sys_crn, sys_crm, sys_op2, t, t2);
```

TBNZ

Test bit and Branch if Nonzero compares the value of a bit in a general-purpose register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.



TBNZ <R><t>, #<imm>, <label>

```
integer t = UInt(Rt);

integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bits(64) offset = SignExtend(imm14:'00', 64);
```

Assembler Symbols

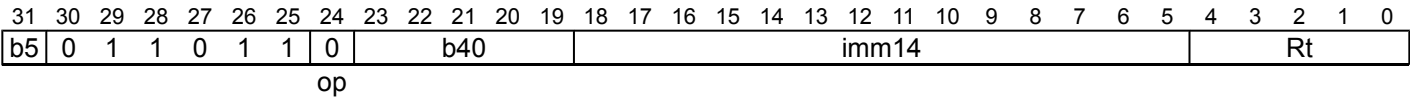
- <R> Is a width specifier, encoded in “b5”:
- | b5 | <R> |
|----|-----|
| 0  | W   |
| 1  | X   |
- In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.
- <t> Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.
- <imm> Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".
- <label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

```
bits(datasize) operand = X[t, datasize];
if operand<bit_pos> == op then
    BranchTo(PC[] + offset, BranchType_DIR, TRUE);
else
    BranchNotTaken(BranchType_DIR, TRUE);
```

TBZ

Test bit and Branch if Zero compares the value of a test bit with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.



TBZ <R><t>, #<imm>, <label>

```
integer t = UInt(Rt);

integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bits(64) offset = SignExtend(imm14:'00', 64);
```

Assembler Symbols

- <R> Is a width specifier, encoded in “b5”:
- | b5 | <R> |
|----|-----|
| 0  | W   |
| 1  | X   |
- In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.
- <t> Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.
- <imm> Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".
- <label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

```
bits(datasize) operand = X[t, datasize];
if operand<bit_pos> == op then
    BranchTo(PC[] + offset, BranchType_DIR, TRUE);
else
    BranchNotTaken(BranchType_DIR, TRUE);
```



# TCANCEL

This instruction exits Transactional state and discards all state modifications that were performed transactionally. Execution continues at the instruction that follows the TSTART instruction of the outer transaction. The destination register of the TSTART instruction of the outer transaction is written with the immediate operand of TCANCEL.

## System (FEAT\_TME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	0	1	1	imm16																0	0	0	0	0

TCANCEL #<imm>

```
if !HaveTME() then UNDEFINED;
boolean retry = (imm16<15> == '1');
bits(15) reason = imm16<14:0>;
```

## Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

## Operation

```
if !IsTMEEnabled() then UNDEFINED;

if TSTATE.depth > 0 then
    FailTransaction(TMFailure_CNCL, retry, FALSE, reason);
```

TCOMMIT

This instruction commits the current transaction. If the current transaction is an outer transaction, then Transactional state is exited, and all state modifications performed transactionally are committed to the architectural state. TCOMMIT takes no inputs and returns no value. Execution of TCOMMIT is UNDEFINED in Non-transactional state.

System  
(FEAT\_TME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	0	1	1	1	1	1	1	1

TCOMMIT

```
if !HaveTME() then UNDEFINED;
```

Operation

```
if !IsTMEEnabled() then UNDEFINED;

if TSTATE.depth == 0 then
    UNDEFINED;

if TSTATE.depth == 1 then
    CommitTransactionalWrites();
    ClearExclusiveLocal(ProcessorID());

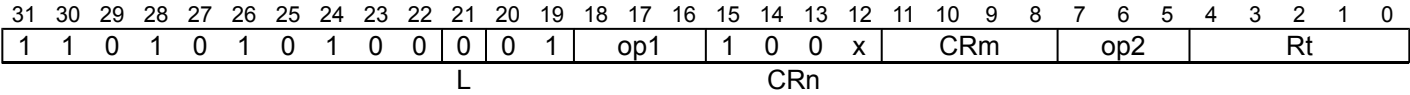
TSTATE.depth = TSTATE.depth - 1;
```

TLBI

TLB Invalidate operation. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions*.

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



```
TLBI <tlbi_op>{, <Xt>}
```

is equivalent to

```
SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

and is the preferred disassembly when `SysOp (op1, CRn, CRm, op2) == Sys_TLBI`.

Assembler Symbols

- <op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- <Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
- <Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- <op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
- <tlbi\_op> Is a TLBI instruction name, as listed for the TLBI system instruction group, encoded in "op1:CRn:CRm:op2":

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature
000	1000	0001	000	VMALLE1OS	FEAT_TLBIOS
000	1000	0001	001	VAE1OS	FEAT_TLBIOS
000	1000	0001	010	ASIDE1OS	FEAT_TLBIOS
000	1000	0001	011	VAAE1OS	FEAT_TLBIOS
000	1000	0001	101	VALE1OS	FEAT_TLBIOS
000	1000	0001	111	VAALE1OS	FEAT_TLBIOS
000	1000	0010	001	RVAE1IS	FEAT_TLBIRANGE
000	1000	0010	011	RVAAE1IS	FEAT_TLBIRANGE
000	1000	0010	101	RVALE1IS	FEAT_TLBIRANGE
000	1000	0010	111	RVAALE1IS	FEAT_TLBIRANGE
000	1000	0011	000	VMALLE1IS	-
000	1000	0011	001	VAE1IS	-
000	1000	0011	010	ASIDE1IS	-
000	1000	0011	011	VAAE1IS	-
000	1000	0011	101	VALE1IS	-
000	1000	0011	111	VAALE1IS	-
000	1000	0101	001	RVAE1OS	FEAT_TLBIRANGE
000	1000	0101	011	RVAAE1OS	FEAT_TLBIRANGE
000	1000	0101	101	RVALE1OS	FEAT_TLBIRANGE
000	1000	0101	111	RVAALE1OS	FEAT_TLBIRANGE
000	1000	0110	001	RVAE1	FEAT_TLBIRANGE
000	1000	0110	011	RVAAE1	FEAT_TLBIRANGE
000	1000	0110	101	RVALE1	FEAT_TLBIRANGE
000	1000	0110	111	RVAALE1	FEAT_TLBIRANGE
000	1000	0111	000	VMALLE1	-
000	1000	0111	001	VAE1	-
000	1000	0111	010	ASIDE1	-
000	1000	0111	011	VAAE1	-
000	1000	0111	101	VALE1	-
000	1000	0111	111	VAALE1	-
000	1001	0001	000	VMALLE1OSNXS	FEAT_XS
000	1001	0001	001	VAE1OSNXS	FEAT_XS
000	1001	0001	010	ASIDE1OSNXS	FEAT_XS
000	1001	0001	011	VAAE1OSNXS	FEAT_XS
000	1001	0001	101	VALE1OSNXS	FEAT_XS
000	1001	0001	111	VAALE1OSNXS	FEAT_XS
000	1001	0010	001	RVAE1ISNXS	FEAT_XS
000	1001	0010	011	RVAAE1ISNXS	FEAT_XS
000	1001	0010	101	RVALE1ISNXS	FEAT_XS
000	1001	0010	111	RVAALE1ISNXS	FEAT_XS
000	1001	0011	000	VMALLE1ISNXS	FEAT_XS
000	1001	0011	001	VAE1ISNXS	FEAT_XS
000	1001	0011	010	ASIDE1ISNXS	FEAT_XS
000	1001	0011	011	VAAE1ISNXS	FEAT_XS
000	1001	0011	101	VALE1ISNXS	FEAT_XS
000	1001	0011	111	VAALE1ISNXS	FEAT_XS
000	1001	0101	001	RVAE1OSNXS	FEAT_XS
000	1001	0101	011	RVAAE1OSNXS	FEAT_XS
000	1001	0101	101	RVALE1OSNXS	FEAT_XS
000	1001	0101	111	RVAALE1OSNXS	FEAT_XS
000	1001	0110	001	RVAE1NXS	FEAT_XS
000	1001	0110	011	RVAAE1NXS	FEAT_XS
000	1001	0110	101	RVALE1NXS	FEAT_XS
000	1001	0110	111	RVAALE1NXS	FEAT_XS
000	1001	0111	000	VMALLE1NXS	FEAT_XS
000	1001	0111	001	VAE1NXS	FEAT_XS
000	1001	0111	010	ASIDE1NXS	FEAT_XS
000	1001	0111	011	VAAE1NXS	FEAT_XS
000	1001	0111	101	VALE1NXS	FEAT_XS
000	1001	0111	111	VAALE1NXS	FEAT_XS
100	1000	0000	001	IPAS2E1IS	-
100	1000	0000	010	RIPAS2E1IS	FEAT_TLBIRANGE
100	1000	0000	101	IPAS2LE1IS	-

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature
100	1000	0000	110	RIPAS2LE1IS	FEAT_TLBIRANGE
100	1000	0001	000	ALLE2OS	FEAT_TLBIOS
100	1000	0001	001	VAE2OS	FEAT_TLBIOS
100	1000	0001	100	ALLE1OS	FEAT_TLBIOS
100	1000	0001	101	VALE2OS	FEAT_TLBIOS
100	1000	0001	110	VMALLS12E1OS	FEAT_TLBIOS
100	1000	0010	001	RVAE2IS	FEAT_TLBIRANGE
100	1000	0010	101	RVALE2IS	FEAT_TLBIRANGE
100	1000	0011	000	ALLE2IS	-
100	1000	0011	001	VAE2IS	-
100	1000	0011	100	ALLE1IS	-
100	1000	0011	101	VALE2IS	-
100	1000	0011	110	VMALLS12E1IS	-
100	1000	0100	000	IPAS2E1OS	FEAT_TLBIOS
100	1000	0100	001	IPAS2E1	-
100	1000	0100	010	RIPAS2E1	FEAT_TLBIRANGE
100	1000	0100	011	RIPAS2E1OS	FEAT_TLBIRANGE
100	1000	0100	100	IPAS2LE1OS	FEAT_TLBIOS
100	1000	0100	101	IPAS2LE1	-
100	1000	0100	110	RIPAS2LE1	FEAT_TLBIRANGE
100	1000	0100	111	RIPAS2LE1OS	FEAT_TLBIRANGE
100	1000	0101	001	RVAE2OS	FEAT_TLBIRANGE
100	1000	0101	101	RVALE2OS	FEAT_TLBIRANGE
100	1000	0110	001	RVAE2	FEAT_TLBIRANGE
100	1000	0110	101	RVALE2	FEAT_TLBIRANGE
100	1000	0111	000	ALLE2	-
100	1000	0111	001	VAE2	-
100	1000	0111	100	ALLE1	-
100	1000	0111	101	VALE2	-
100	1000	0111	110	VMALLS12E1	-
100	1001	0000	001	IPAS2E1ISNXS	FEAT_XS
100	1001	0000	010	RIPAS2E1ISNXS	FEAT_XS
100	1001	0000	101	IPAS2LE1ISNXS	FEAT_XS
100	1001	0000	110	RIPAS2LE1ISNXS	FEAT_XS
100	1001	0001	000	ALLE2OSNXS	FEAT_XS
100	1001	0001	001	VAE2OSNXS	FEAT_XS
100	1001	0001	100	ALLE1OSNXS	FEAT_XS
100	1001	0001	101	VALE2OSNXS	FEAT_XS
100	1001	0001	110	VMALLS12E1OSNXS	FEAT_XS
100	1001	0010	001	RVAE2ISNXS	FEAT_XS
100	1001	0010	101	RVALE2ISNXS	FEAT_XS
100	1001	0011	000	ALLE2ISNXS	FEAT_XS
100	1001	0011	001	VAE2ISNXS	FEAT_XS
100	1001	0011	100	ALLE1ISNXS	FEAT_XS
100	1001	0011	101	VALE2ISNXS	FEAT_XS
100	1001	0011	110	VMALLS12E1ISNXS	FEAT_XS
100	1001	0100	000	IPAS2E1OSNXS	FEAT_XS
100	1001	0100	001	IPAS2E1NXS	FEAT_XS
100	1001	0100	010	RIPAS2E1NXS	FEAT_XS
100	1001	0100	011	RIPAS2E1OSNXS	FEAT_XS
100	1001	0100	100	IPAS2LE1OSNXS	FEAT_XS
100	1001	0100	101	IPAS2LE1NXS	FEAT_XS
100	1001	0100	110	RIPAS2LE1NXS	FEAT_XS
100	1001	0100	111	RIPAS2LE1OSNXS	FEAT_XS
100	1001	0101	001	RVAE2OSNXS	FEAT_XS
100	1001	0101	101	RVALE2OSNXS	FEAT_XS
100	1001	0110	001	RVAE2NXS	FEAT_XS
100	1001	0110	101	RVALE2NXS	FEAT_XS
100	1001	0111	000	ALLE2NXS	FEAT_XS
100	1001	0111	001	VAE2NXS	FEAT_XS
100	1001	0111	100	ALLE1NXS	FEAT_XS
100	1001	0111	101	VALE2NXS	FEAT_XS
100	1001	0111	110	VMALLS12E1NXS	FEAT_XS
110	1000	0001	000	ALLE3OS	FEAT_TLBIOS

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature
110	1000	0001	001	VAE3OS	FEAT_TLBIOS
110	1000	0001	100	PAALLOS	FEAT_RME
110	1000	0001	101	VALE3OS	FEAT_TLBIOS
110	1000	0010	001	RVAE3IS	FEAT_TLBIRANGE
110	1000	0010	101	RVALE3IS	FEAT_TLBIRANGE
110	1000	0011	000	ALLE3IS	-
110	1000	0011	001	VAE3IS	-
110	1000	0011	101	VALE3IS	-
110	1000	0100	011	RPAOS	FEAT_RME
110	1000	0100	111	RPALOS	FEAT_RME
110	1000	0101	001	RVAE3OS	FEAT_TLBIRANGE
110	1000	0101	101	RVALE3OS	FEAT_TLBIRANGE
110	1000	0110	001	RVAE3	FEAT_TLBIRANGE
110	1000	0110	101	RVALE3	FEAT_TLBIRANGE
110	1000	0111	000	ALLE3	-
110	1000	0111	001	VAE3	-
110	1000	0111	100	PAALL	FEAT_RME
110	1000	0111	101	VALE3	-
110	1001	0001	000	ALLE3OSNXS	FEAT_XS
110	1001	0001	001	VAE3OSNXS	FEAT_XS
110	1001	0001	101	VALE3OSNXS	FEAT_XS
110	1001	0010	001	RVAE3ISNXS	FEAT_XS
110	1001	0010	101	RVALE3ISNXS	FEAT_XS
110	1001	0011	000	ALLE3ISNXS	FEAT_XS
110	1001	0011	001	VAE3ISNXS	FEAT_XS
110	1001	0011	101	VALE3ISNXS	FEAT_XS
110	1001	0101	001	RVAE3OSNXS	FEAT_XS
110	1001	0101	101	RVALE3OSNXS	FEAT_XS
110	1001	0110	001	RVAE3NXS	FEAT_XS
110	1001	0110	101	RVALE3NXS	FEAT_XS
110	1001	0111	000	ALLE3NXS	FEAT_XS
110	1001	0111	001	VAE3NXS	FEAT_XS
110	1001	0111	101	VALE3NXS	FEAT_XS

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

## Operation

The description of [SYS](#) gives the operational pseudocode for this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# TLBIP

TLB Invalidate Pair operation.

This is an alias of [SYSP](#). This means:

- The encodings in this description are named to match the encodings of [SYSP](#).
- The description of [SYSP](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

## System (FEAT\_D128)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	1	0	0	1	op1			1	0	0	x	CRm			op2			Rt					
L											CRn																				

TLBIP <tlbip\_op>{, <Xt1>, <Xt2>}

is equivalent to

[SYSP](#) #<op1>, <Cn>, <Cm>, #<op2>{, <Xt1>, <Xt2>}

and is the preferred disassembly when SysOp (op1, CRn, CRm, op2) == Sys\_TLBIP.

## Assembler Symbols

- <op1>

Is a 3-bit unsigned immediate, in the range 0 to 6, encoded in the "op1" field.
- <Cn>

Is a name 'Cn', with 'n' in the range 8 to 9, encoded in the "CRn" field.
- <Cm>

Is a name 'Cm', with 'm' in the range 0 to 7, encoded in the "CRm" field.
- <op2>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
- <tlbip\_op>

Is a TLBIP instruction name, as listed for the TLBIP system pair instruction group, encoded in "op1:CRn:CRm:op2":

op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature
000	1000	0001	001	VAE1OS	FEAT_D128
000	1000	0001	011	VAAE1OS	FEAT_D128
000	1000	0001	101	VALE1OS	FEAT_D128
000	1000	0001	111	VAALE1OS	FEAT_D128
000	1000	0010	001	RVAE1IS	FEAT_D128
000	1000	0010	011	RVAAE1IS	FEAT_D128
000	1000	0010	101	RVALE1IS	FEAT_D128
000	1000	0010	111	RVAALE1IS	FEAT_D128
000	1000	0011	001	VAE1IS	FEAT_D128
000	1000	0011	011	VAAE1IS	FEAT_D128
000	1000	0011	101	VALE1IS	FEAT_D128
000	1000	0011	111	VAALE1IS	FEAT_D128
000	1000	0101	001	RVAE1OS	FEAT_D128
000	1000	0101	011	RVAAE1OS	FEAT_D128
000	1000	0101	101	RVALE1OS	FEAT_D128
000	1000	0101	111	RVAALE1OS	FEAT_D128
000	1000	0110	001	RVAE1	FEAT_D128
000	1000	0110	011	RVAAE1	FEAT_D128
000	1000	0110	101	RVALE1	FEAT_D128
000	1000	0110	111	RVAALE1	FEAT_D128
000	1000	0111	001	VAE1	FEAT_D128
000	1000	0111	011	VAAE1	FEAT_D128
000	1000	0111	101	VALE1	FEAT_D128
000	1000	0111	111	VAALE1	FEAT_D128
000	1001	0001	001	VAE1OSNXS	FEAT_D128
000	1001	0001	011	VAAE1OSNXS	FEAT_D128
000	1001	0001	101	VALE1OSNXS	FEAT_D128
000	1001	0001	111	VAALE1OSNXS	FEAT_D128
000	1001	0010	001	RVAE1ISNXS	FEAT_D128
000	1001	0010	011	RVAAE1ISNXS	FEAT_D128
000	1001	0010	101	RVALE1ISNXS	FEAT_D128
000	1001	0010	111	RVAALE1ISNXS	FEAT_D128
000	1001	0011	001	VAE1ISNXS	FEAT_D128
000	1001	0011	011	VAAE1ISNXS	FEAT_D128
000	1001	0011	101	VALE1ISNXS	FEAT_D128
000	1001	0011	111	VAALE1ISNXS	FEAT_D128
000	1001	0101	001	RVAE1OSNXS	FEAT_D128
000	1001	0101	011	RVAAE1OSNXS	FEAT_D128
000	1001	0101	101	RVALE1OSNXS	FEAT_D128
000	1001	0101	111	RVAALE1OSNXS	FEAT_D128
000	1001	0110	001	RVAE1NXS	FEAT_D128
000	1001	0110	011	RVAAE1NXS	FEAT_D128
000	1001	0110	101	RVALE1NXS	FEAT_D128
000	1001	0110	111	RVAALE1NXS	FEAT_D128
000	1001	0111	001	VAE1NXS	FEAT_D128
000	1001	0111	011	VAAE1NXS	FEAT_D128
000	1001	0111	101	VALE1NXS	FEAT_D128
000	1001	0111	111	VAALE1NXS	FEAT_D128
100	1000	0000	001	IPAS2E1IS	FEAT_D128
100	1000	0000	010	RIPAS2E1IS	FEAT_D128
100	1000	0000	101	IPAS2LE1IS	FEAT_D128
100	1000	0000	110	RIPAS2LE1IS	FEAT_D128
100	1000	0001	001	VAE2OS	FEAT_D128
100	1000	0001	101	VALE2OS	FEAT_D128
100	1000	0010	001	RVAE2IS	FEAT_D128
100	1000	0010	101	RVALE2IS	FEAT_D128
100	1000	0011	001	VAE2IS	FEAT_D128
100	1000	0011	101	VALE2IS	FEAT_D128
100	1000	0100	000	IPAS2E1OS	FEAT_D128
100	1000	0100	001	IPAS2E1	FEAT_D128
100	1000	0100	010	RIPAS2E1	FEAT_D128
100	1000	0100	011	RIPAS2E1OS	FEAT_D128
100	1000	0100	100	IPAS2LE1OS	FEAT_D128



op1	CRn	CRm	op2	<tlbip_op>	Architectural Feature
100	1000	0100	101	IPAS2LE1	FEAT_D128
100	1000	0100	110	RIPAS2LE1	FEAT_D128
100	1000	0100	111	RIPAS2LE1OS	FEAT_D128
100	1000	0101	001	RVAE2OS	FEAT_D128
100	1000	0101	101	RVALE2OS	FEAT_D128
100	1000	0110	001	RVAE2	FEAT_D128
100	1000	0110	101	RVALE2	FEAT_D128
100	1000	0111	001	VAE2	FEAT_D128
100	1000	0111	101	VALE2	FEAT_D128
100	1001	0000	001	IPAS2E1ISNXS	FEAT_D128
100	1001	0000	010	RIPAS2E1ISNXS	FEAT_D128
100	1001	0000	101	IPAS2LE1ISNXS	FEAT_D128
100	1001	0000	110	RIPAS2LE1ISNXS	FEAT_D128
100	1001	0001	001	VAE2OSNXS	FEAT_D128
100	1001	0001	101	VALE2OSNXS	FEAT_D128
100	1001	0010	001	RVAE2ISNXS	FEAT_D128
100	1001	0010	101	RVALE2ISNXS	FEAT_D128
100	1001	0011	001	VAE2ISNXS	FEAT_D128
100	1001	0011	101	VALE2ISNXS	FEAT_D128
100	1001	0100	000	IPAS2E1OSNXS	FEAT_D128
100	1001	0100	001	IPAS2E1NXS	FEAT_D128
100	1001	0100	010	RIPAS2E1NXS	FEAT_D128
100	1001	0100	011	RIPAS2E1OSNXS	FEAT_D128
100	1001	0100	100	IPAS2LE1OSNXS	FEAT_D128
100	1001	0100	101	IPAS2LE1NXS	FEAT_D128
100	1001	0100	110	RIPAS2LE1NXS	FEAT_D128
100	1001	0100	111	RIPAS2LE1OSNXS	FEAT_D128
100	1001	0101	001	RVAE2OSNXS	FEAT_D128
100	1001	0101	101	RVALE2OSNXS	FEAT_D128
100	1001	0110	001	RVAE2NXS	FEAT_D128
100	1001	0110	101	RVALE2NXS	FEAT_D128
100	1001	0111	001	VAE2NXS	FEAT_D128
100	1001	0111	101	VALE2NXS	FEAT_D128
110	1000	0001	001	VAE3OS	FEAT_D128
110	1000	0001	101	VALE3OS	FEAT_D128
110	1000	0010	001	RVAE3IS	FEAT_D128
110	1000	0010	101	RVALE3IS	FEAT_D128
110	1000	0011	001	VAE3IS	FEAT_D128
110	1000	0011	101	VALE3IS	FEAT_D128
110	1000	0101	001	RVAE3OS	FEAT_D128
110	1000	0101	101	RVALE3OS	FEAT_D128
110	1000	0110	001	RVAE3	FEAT_D128
110	1000	0110	101	RVALE3	FEAT_D128
110	1000	0111	001	VAE3	FEAT_D128
110	1000	0111	101	VALE3	FEAT_D128
110	1001	0001	001	VAE3OSNXS	FEAT_D128
110	1001	0001	101	VALE3OSNXS	FEAT_D128
110	1001	0010	001	RVAE3ISNXS	FEAT_D128
110	1001	0010	101	RVALE3ISNXS	FEAT_D128
110	1001	0011	001	VAE3ISNXS	FEAT_D128
110	1001	0011	101	VALE3ISNXS	FEAT_D128
110	1001	0101	001	RVAE3OSNXS	FEAT_D128
110	1001	0101	101	RVALE3OSNXS	FEAT_D128
110	1001	0110	001	RVAE3NXS	FEAT_D128
110	1001	0110	101	RVALE3NXS	FEAT_D128
110	1001	0111	001	VAE3NXS	FEAT_D128
110	1001	0111	101	VALE3NXS	FEAT_D128

<Xt1> Is the 64-bit name of the first optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second optional general-purpose source register, defaulting to '11111', encoded as "Rt" + 1. Defaults to '11111' if "Rt" = '11111'.

## Operation

The description of [SYSP](#) gives the operational pseudocode for this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TRCIT

Trace Instrumentation generates an instrumentation trace packet that contains the value of the provided register.

This is an alias of [SYS](#). This means:

- The encodings in this description are named to match the encodings of [SYS](#).
- The description of [SYS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System  
(FEAT\_ITE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0	1	1	1	0	0	1	0	1	1	1	Rt				
L										op1				CRn				CRm				op2									

TRCIT [<Xt>](#)

is equivalent to

[SYS](#) #3, C7, C2, #7, [<Xt>](#)

and is always the preferred disassembly.

Assembler Symbols

[<Xt>](#) Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

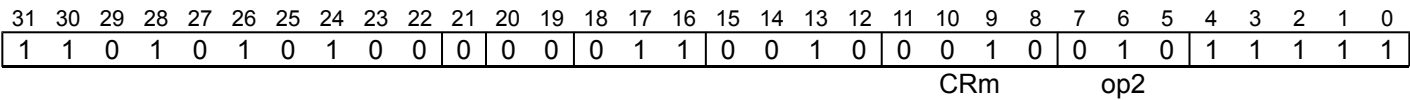
Operation

The description of [SYS](#) gives the operational pseudocode for this instruction.

TSB CSYNC

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions, see *Trace Synchronization Buffer (TSB CSYNC)*.  
If *FEAT\_TRF* is not implemented, this instruction executes as a NOP.

System  
(FEAT\_TRF)



TSB CSYNC

```
if !HaveSelfHostedTrace() then EndOfInstruction();
```

Operation

```
TraceSynchronizationBarrier();
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42  
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TST (immediate)

Test bits (immediate), setting the condition flags and discarding the result

: Rn AND imm.

This is an alias of [ANDS \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [ANDS \(immediate\)](#).
- The description of [ANDS \(immediate\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
sf	1	1	1	0	0	1	0	0	N	immr						imms						Rn						1	1	1	1	1
opc										Rd																						

32-bit (sf == 0 && N == 0)

TST <Wn>, #<imm>

is equivalent to

ANDS WZR, <Wn>, #<imm>

and is always the preferred disassembly.

64-bit (sf == 1)

TST <Xn>, #<imm>

is equivalent to

ANDS XZR, <Xn>, #<imm>

and is always the preferred disassembly.

Assembler Symbols

- <Wn>

Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xn>

Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <imm>

For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".  
For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Operation

The description of [ANDS \(immediate\)](#) gives the operational pseudocode for this instruction.

TST (shifted register)

Test (shifted register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This is an alias of [ANDS \(shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [ANDS \(shifted register\)](#).
- The description of [ANDS \(shifted register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
sf	1	1	0	1	0	1	0	shift	0	Rm						imm6						Rn						1	1	1	1	1
opc								N												Rd												

32-bit (sf == 0)

TST <Wn>, <Wm>{, <shift> #<amount>}

is equivalent to

ANDS WZR, <Wn>, <Wm>{, <shift> #<amount>}

and is always the preferred disassembly.

64-bit (sf == 1)

TST <Xn>, <Xm>{, <shift> #<amount>}

is equivalent to

ANDS XZR, <Xn>, <Xm>{, <shift> #<amount>}

and is always the preferred disassembly.

Assembler Symbols

<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.										
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.										
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.										
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.										
<shift>	Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift": <table border="1"><thead><tr><th>shift</th><th>&lt;shift&gt;</th></tr></thead><tbody><tr><td>00</td><td>LSL</td></tr><tr><td>01</td><td>LSR</td></tr><tr><td>10</td><td>ASR</td></tr><tr><td>11</td><td>ROR</td></tr></tbody></table>	shift	<shift>	00	LSL	01	LSR	10	ASR	11	ROR
shift	<shift>										
00	LSL										
01	LSR										
10	ASR										
11	ROR										
<amount>	For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,										

Operation

The description of [ANDS \(shifted register\)](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# TSTART

This instruction starts a new transaction. If the transaction started successfully, the destination register is set to zero. If the transaction failed or was canceled, then all state modifications that were performed transactionally are discarded and the destination register is written with a nonzero value that encodes the cause of the failure.

## System (FEAT\_TME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	1	0	0	0	1	1	0	0	1	1	0	0	0	0	0	1	1	Rt				

TSTART <Xt>

```
if !HaveTME() then UNDEFINED;
integer t = UInt(Rt);
```

## Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.



## Operation

```
if !IsTMEEnabled() then UNDEFINED;

boolean IsEL1Regime;
bit tme;
bit tmt;
case PSTATE.EL of
  when EL0
    IsEL1Regime = S1TranslationRegime() == EL1;
    if IsEL1Regime then
      tme = SCTLR_EL1.TME0;
      tmt = SCTLR_EL1.TMT0;
    else
      tme = SCTLR_EL2.TME0;
      tmt = SCTLR_EL2.TMT0;
  when EL1
    tme = SCTLR_EL1.TME;
    tmt = SCTLR_EL1.TMT;
  when EL2
    tme = SCTLR_EL2.TME;
    tmt = SCTLR_EL2.TMT;
  when EL3
    tme = SCTLR_EL3.TME;
    tmt = SCTLR_EL3.TMT;
  otherwise
    Unreachable();

boolean enable = tme == '1';
boolean trivial = tmt == '1';

if !enable then
  TransactionStartTrap(t);
elseif trivial then
  TSTATE.nPC = NextInstrAddr(64);
  TSTATE.Rt = t;
  FailTransaction(TMFailure_TRIVIAL, FALSE);
elseif HaveSME() && PSTATE.SM == '1' then
  FailTransaction(TMFailure_ERR, FALSE);
elseif TSTATE.depth == 255 then
  FailTransaction(TMFailure_NEST, FALSE);
elseif TSTATE.depth == 0 then
  TSTATE.nPC = NextInstrAddr(64);
  TSTATE.Rt = t;
  ClearExclusiveLocal(ProcessorID());
  TakeTransactionCheckpoint();
  StartTrackingTransactionalReadsWrites();

TSTATE.depth = TSTATE.depth + 1;
X[t, 64] = Zeros(64);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# TTEST

This instruction writes the depth of the transaction to the destination register, or the value 0 otherwise.

## System (FEAT\_TME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	1	0	0	0	1	1	0	0	1	1	0	0	0	1	0	1	1				Rt	

TTEST <Xt>

```
if !HaveTME() then UNDEFINED;
integer t = UInt(Rt);
```

## Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.

## Operation

```
if !IsTMEEnabled() then UNDEFINED;
X[t, 64] = (TSTATE.depth)<63:0>;
```

# UBFIZ

Unsigned Bitfield Insert in Zeros copies a bitfield of <width> bits from the least significant bits of the source register to bit position <lsb> of the destination register, setting the destination bits above and below the bitfield to zero.

This is an alias of [UBFM](#). This means:

- The encodings in this description are named to match the encodings of [UBFM](#).
- The description of [UBFM](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
sf		1		0		1		0		0		1		1		0		N		immr						imms						Rn				Rd			
opc																																							

## 32-bit (sf == 0 && N == 0)

```
UBFIZ <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

## 64-bit (sf == 1 && N == 1)

```
UBFIZ <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
<width>	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

## Operation

The description of [UBFM](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



UBFM

Unsigned Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If <imms> is greater than or equal to <immr>, this copies a bitfield of (<imms>-<immr>+1) bits starting from bit position <immr> in the source register to the least significant bits of the destination register.

If <imms> is less than <immr>, this copies a bitfield of (<imms>+1) bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases the destination bits below and above the bitfield are set to zero.

This instruction is used by the aliases [LSL \(immediate\)](#), [LSR \(immediate\)](#), [UBFIZ](#), [UBFX](#), [UXTB](#), and [UXTH](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
sf		1		0		1		0		0		1		1		0		N		immr						imms						Rn				Rd			

opc

32-bit (sf == 0 && N == 0)

```
UBFM <Wd>, <Wn>, #<immr>, #<imms>
```

64-bit (sf == 1 && N == 1)

```
UBFM <Xd>, <Xn>, #<immr>, #<imms>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

integer r;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;

r = UInt(immr);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE, datasize);
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <immr> For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.  
For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.
- <imms> For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.  
For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

Alias	Of variant	Is preferred when
<a href="#">LSL (immediate)</a>	32-bit	imms != '011111' && imms + 1 == immr
<a href="#">LSL (immediate)</a>	64-bit	imms != '111111' && imms + 1 == immr
<a href="#">LSR (immediate)</a>	32-bit	imms == '011111'
<a href="#">LSR (immediate)</a>	64-bit	imms == '111111'

Alias	Of variant	Is preferred when
<a href="#">UBFIZ</a>		<a href="#">UInt</a> (imms) < <a href="#">UInt</a> (immr)
<a href="#">UBFX</a>		<a href="#">BFXPreferred</a> (sf, opc<1>, imms, immr)
<a href="#">UXTB</a>		immr == '000000' && imms == '000111'
<a href="#">UXTH</a>		immr == '000000' && imms == '001111'

## Operation

```
bits(datasize) src = X[n, datasize];

// perform bitfield move on low bits
bits(datasize) bot = ROR(src, r) AND wmask;

// combine extension bits and result bits
X[d, datasize] = bot AND tmask;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# UBFX

Unsigned Bitfield Extract copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to zero.

This is an alias of [UBFM](#). This means:

- The encodings in this description are named to match the encodings of [UBFM](#).
  - The description of [UBFM](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.
- |     |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |      |    |   |   |   |   |      |   |   |   |   |   |    |  |  |  |  |  |    |  |  |  |  |  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|------|----|---|---|---|---|------|---|---|---|---|---|----|--|--|--|--|--|----|--|--|--|--|--|
| 31  | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11   | 10 | 9 | 8 | 7 | 6 | 5    | 4 | 3 | 2 | 1 | 0 |    |  |  |  |  |  |    |  |  |  |  |  |
| sf  |    | 1  |    | 0  |    | 1  |    | 0  |    | 0  |    | 1  |    | 1  |    | 0  |    | N  |    | immr |    |   |   |   |   | imms |   |   |   |   |   | Rn |  |  |  |  |  | Rd |  |  |  |  |  |
| opc |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |      |    |   |   |   |   |      |   |   |   |   |   |    |  |  |  |  |  |    |  |  |  |  |  |

## 32-bit (sf == 0 && N == 0)

UBFX <Wd>, <Wn>, #<lsb>, #<width>

is equivalent to

UBFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

## 64-bit (sf == 1 && N == 1)

UBFX <Xd>, <Xn>, #<lsb>, #<width>

is equivalent to

UBFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

## Assembler Symbols

- |         |  |
|---------|--|
| <Wd>    | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.   |
| <Wn>    | Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.  |
| <Xd>    | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.   |
| <Xn>    | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.  |
| <lsb>   | For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.<br>For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63. |
| <width> | For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.<br>For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.                                   |

## Operation

The description of [UBFM](#) gives the operational pseudocode for this instruction.

## Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.





# UDF

Permanently Undefined generates an Undefined Instruction exception (ESR\_ELx.EC = 0b0000000). The encodings for UDF used in this section are defined as permanently UNDEFINED.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	imm16															

UDF #<imm>

```
// The imm16 field is ignored by hardware.  
UNDEFINED;
```

## Assembler Symbols

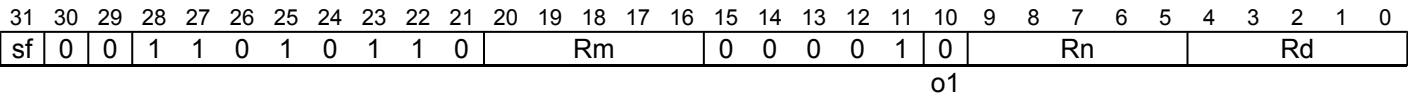
<imm> is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field. The PE ignores the value of this constant.

## Operation

```
// No operation.
```

UDIV

Unsigned Divide divides an unsigned integer register value by another unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.



32-bit (sf == 0)

```
UDIV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
UDIV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
integer result;

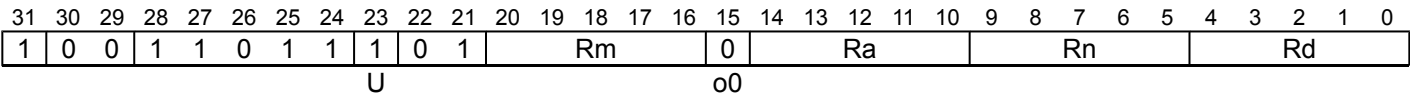
if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, TRUE)) / Real(Int(operand2, TRUE)));

X[d, datasize] = result<datasize-1:0>;
```

UMADDL

Unsigned Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias [UMULL](#).



UMADDL <Xd>, <Wn>, <Wm>, <Xa>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- <Xa> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
<a href="#">UMULL</a>	Ra == '11111'

Operation

```
bits(32) operand1 = X[n, 32];
bits(32) operand2 = X[m, 32];
bits(64) operand3 = X[a, 64];

integer result;

result = Int(operand3, TRUE) + (Int(operand1, TRUE) * Int(operand2, TRUE));

X[d, 64] = result<63:0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

UMAX (immediate)

Unsigned Maximum (immediate) determines the unsigned maximum of the source register value and immediate, and writes the result to the destination register.

Integer  
(FEAT\_CSSC)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	0	0	0	1	1	1	0	0	0	1	imm8								Rn				Rd					

32-bit (sf == 0)

```
UMAX <Wd>, <Wn>, #<uimm>
```

64-bit (sf == 1)

```
UMAX <Xd>, <Xn>, #<uimm>
```

```
if !HaveCSSC() then UNDEFINED;
integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = UInt(imm8);
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <uimm> Is an unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
integer result = Max(UInt(operand1), imm);
X[d, datasize] = result<datasize-1:0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

UMAX (register)

Unsigned Maximum (register) determines the unsigned maximum of the two source register values and writes the result to the destination register.

Integer  
(FEAT\_CSSC)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	1	1	0	0	1	Rn					Rd				

32-bit (sf == 0)

UMAX <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

UMAX <Xd>, <Xn>, <Xm>

```
if !HaveCSSC() then UNDEFINED;
integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
integer result = Max(UInt(operand1), UInt(operand2));
X[d, datasize] = result<datasize-1:0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMIN (immediate)

Unsigned Minimum (immediate) determines the unsigned minimum of the source register value and immediate, and writes the result to the destination register.

Integer  
(FEAT\_CSSC)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	0	0	0	1	1	1	0	0	1	1	imm8								Rn				Rd					

32-bit (sf == 0)

UMIN <Wd>, <Wn>, #<uimm>

64-bit (sf == 1)

UMIN <Xd>, <Xn>, #<uimm>

```
if !HaveCSSC() then UNDEFINED;
integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = UInt(imm8);
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <uimm> Is an unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
integer result = Min(UInt(operand1), imm);
X[d, datasize] = result<datasize-1:0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

UMIN (register)

Unsigned Minimum (register) determines the unsigned minimum of the two source register values and writes the result to the destination register.

Integer  
(FEAT\_CSSC)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	0	1	0	1	1	0	Rm					0	1	1	0	1	1	Rn					Rd				

32-bit (sf == 0)

UMIN <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

UMIN <Xd>, <Xn>, <Xm>

```
if !HaveCSSC() then UNDEFINED;
integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
integer result = Min(UInt(operand1), UInt(operand2));
X[d, datasize] = result<datasize-1:0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMNEGL

Unsigned Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of [UMSUBL](#). This means:

- The encodings in this description are named to match the encodings of [UMSUBL](#).
- The description of [UMSUBL](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	1	1	0	1	Rm					1	1	1	1	1	1	Rn					Rd				
U											o0					Ra															

UMNEGL <Xd>, <Wn>, <Wm>

is equivalent to

[UMSUBL](#) <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of [UMSUBL](#) gives the operational pseudocode for this instruction.

Operational information

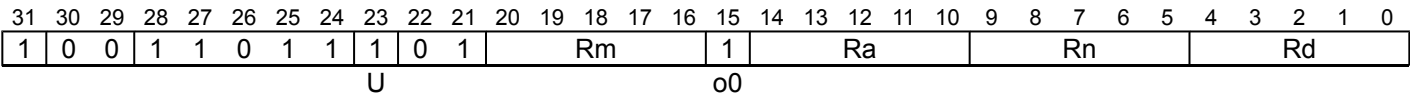
- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



UMSUBL

Unsigned Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias [UMNEGL](#).



UMSUBL <Xd>, <Wn>, <Wm>, <Xa>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- <Xa> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
<a href="#">UMNEGL</a>	Ra == '11111'

Operation

```
bits(32) operand1 = X[n, 32];
bits(32) operand2 = X[m, 32];
bits(64) operand3 = X[a, 64];

integer result;

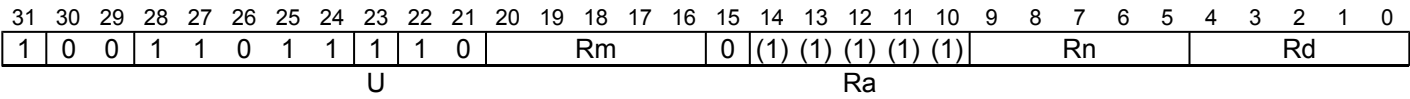
result = Int(operand3, TRUE) - (Int(operand1, TRUE) * Int(operand2, TRUE));
X[d, 64] = result<63:0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

# UMULH

Unsigned Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.



UMULH <Xd>, <Xn>, <Xm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

## Assembler Symbols

- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

## Operation

```
bits(64) operand1 = X[n, 64];
bits(64) operand2 = X[m, 64];

integer result;

result = Int(operand1, TRUE) * Int(operand2, TRUE);

X[d, 64] = result<127:64>;
```

## Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# UMULL

Unsigned Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of [UMADDL](#). This means:

- The encodings in this description are named to match the encodings of [UMADDL](#).
  - The description of [UMADDL](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.
- |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|----|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | Rm |    |    |    |    |    | 0  | 1  | 1  | 1  | 1  | Rn |   |   |   |   |   | Rd |   |   |   |
| U  |    |    |    |    |    |    |    |    |    |    | o0 |    |    |    |    |    | Ra |    |    |    |    |    |   |   |   |   |   |    |   |   |   |

UMULL <Xd>, <Wn>, <Wm>

is equivalent to

[UMADDL](#) <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

## Assembler Symbols

- |      |  |
|------|--|
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.                           |
| <Wn> | Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Wm> | Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.  |

## Operation

The description of [UMADDL](#) gives the operational pseudocode for this instruction.

## Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This is an alias of [UBFM](#). This means:

- The encodings in this description are named to match the encodings of [UBFM](#).
- The description of [UBFM](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	Rn						Rd					
sf			opc									N			immr						imms												

32-bit

UXTB <Wd>, <Wn>

is equivalent to

[UBFM](#) <Wd>, <Wn>, #0, #7

and is always the preferred disassembly.

Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>

Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of [UBFM](#) gives the operational pseudocode for this instruction.

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

UXTH

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This is an alias of [UBFM](#). This means:

- The encodings in this description are named to match the encodings of [UBFM](#).
- The description of [UBFM](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	Rn				Rd					
sf			opc								N		immr				imms														

32-bit

UXTH <Wd>, <Wn>

is equivalent to

[UBFM](#) <Wd>, <Wn>, #0, #15

and is always the preferred disassembly.

Assembler Symbols

- <Wd>

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn>

Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of [UBFM](#) gives the operational pseudocode for this instruction.

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

WFE

Wait For Event is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the SEV instruction on any PE in the multiprocessor system. For more information, see [Wait For Event mechanism and Send event](#).

As described in [Wait For Event mechanism and Send event](#), the execution of a WFE instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	1	0	1	1	1	1	1
																CRm				op2											

WFE

// Empty.

Operation

```
integer localtimeout = 1 << 64;      // No local timeout event is generated
Hint_WFE(localtimeout, WFxType_WFE);
```

WFET

Wait For Event with Timeout is a hint instruction that indicates that the PE can enter a low-power state and remain there until either a local timeout event or a wakeup event occurs. Wakeup events include the event signaled as a result of executing the SEV instruction on any PE in the multiprocessor system. For more information, see [Wait For Event mechanism and Send event](#).

As described in [Wait For Event mechanism and Send event](#), the execution of a WFET instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

System  
(FEAT\_WFxT)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0				Rd

WFET <Xt>

```
if !HaveFeatWFxT() then UNDEFINED;

integer d = UInt(Rd);
```

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rd" field.

Operation

```
integer localtimeout = UInt(X[d, 64]);

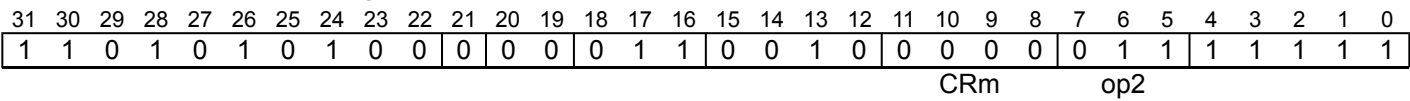
if Halted() && ConstrainUnpredictableBool(Unpredictable WFxTDEBUG) then
    EndOfInstruction();

Hint_WFE(localtimeout, WFxType_WFET);
```

WFI

Wait For Interrupt is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see [Wait For Interrupt](#).

As described in [Wait For Interrupt](#), the execution of a WFI instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.



WFI

```
// Empty.
```

Operation

```
integer localtimeout = 1 << 64;    // No local timeout event is generated
Hint\_WFI(localtimeout, WfxType\_WFI);
```



WFIT

Wait For Interrupt with Timeout is a hint instruction that indicates that the PE can enter a low-power state and remain there until either a local timeout event or a wakeup event occurs. For more information, see [Wait For Interrupt](#). As described in [Wait For Interrupt](#), the execution of a `WFIT` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

System  
(FEAT\_WFxT)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	1					Rd

`WFIT` `<Xt>`

```
if !HaveFeatWFxT() then UNDEFINED;

integer d = UInt(Rd);
```

Assembler Symbols

`<Xt>` Is the 64-bit name of the general-purpose source register, encoded in the "Rd" field.

Operation

```
integer localtimeout = UInt(X[d, 64]);

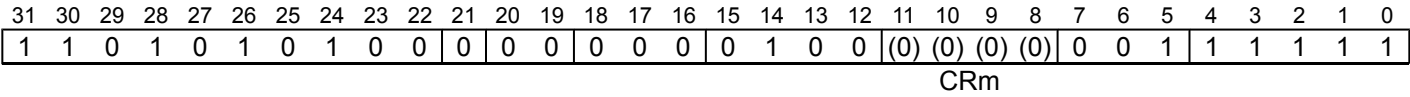
if Halted() && ConstrainUnpredictableBool(Unpredictable_WFxTDEBUG) then
    EndOfInstruction();

Hint_WFI(localtimeout, WFxType_WFIT);
```

XAFLAG

Convert floating-point condition flags from external format to Arm format. This instruction converts the state of the PSTATE.{N,Z,C,V} flags from an alternative representation required by some software to a form representing the result of an Arm floating-point scalar compare instruction.

System  
(FEAT\_FlagM2)



XAFLAG

```
if !HaveFlagFormatExt() then UNDEFINED;
```

Operation

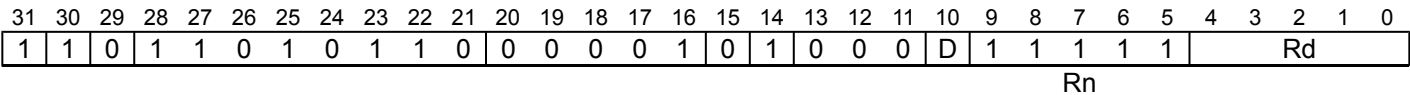
```
bit n = NOT(PSTATE.C) AND NOT(PSTATE.Z);
bit z = PSTATE.Z AND PSTATE.C;
bit c = PSTATE.C OR PSTATE.Z;
bit v = NOT(PSTATE.C) AND PSTATE.Z;

PSTATE.N = n;
PSTATE.Z = z;
PSTATE.C = c;
PSTATE.V = v;
```

XPACD, XPACI, XPACLR

Strip Pointer Authentication Code. This instruction removes the pointer authentication code from an address. The address is in the specified general-purpose register for `XPACI` and `XPACD`, and is in `LR` for `XPACLR`. The `XPACD` instruction is used for data addresses, and `XPACI` and `XPACLR` are used for instruction addresses. It has encodings from 2 classes: [Integer](#) and [System](#)

Integer  
(FEAT\_PAuth)



XPACD (D == 1)

```
XPACD <Xd>
```

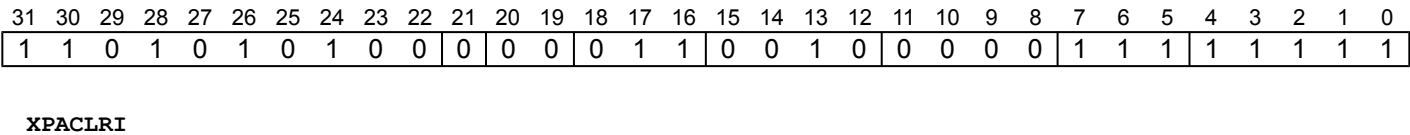
XPACI (D == 0)

```
XPACI <Xd>

boolean data = (D == '1');
integer d = UInt(Rd);

if !HavePACExt() then
    UNDEFINED;
```

System  
(FEAT\_PAuth)



```
integer d = 30;
boolean data = FALSE;
```

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Operation

```
if HavePACExt() then
    X[d, 64] = Strip(X[d, 64], data);
```

YIELD

YIELD is a hint instruction. Software with a multithreading capability can use a YIELD instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction, see [The YIELD instruction](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1
																CRm				op2											

YIELD

// Empty.

Operation

[Hint\\_Yield\(\)](#);

## A64 -- SIMD and Floating-point Instructions (alphabetic order)

[ABS](#): Absolute value (vector).

[ADD \(vector\)](#): Add (vector).

[ADDHN, ADDHN2](#): Add returning High Narrow.

[ADDP \(scalar\)](#): Add Pair of elements (scalar).

[ADDP \(vector\)](#): Add Pairwise (vector).

[ADDV](#): Add across Vector.

[AESD](#): AES single round decryption.

[AESE](#): AES single round encryption.

[AESIMC](#): AES inverse mix columns.

[AESMC](#): AES mix columns.

[AND \(vector\)](#): Bitwise AND (vector).

[BCAX](#): Bit Clear and exclusive-OR.

[BFCVT](#): Floating-point convert from single-precision to BFloat16 format (scalar).

[BFCVTN, BFCVTN2](#): Floating-point convert from single-precision to BFloat16 format (vector).

[BFDOT \(by element\)](#): BFloat16 floating-point dot product (vector, by element).

[BFDOT \(vector\)](#): BFloat16 floating-point dot product (vector).

[BFMLALB, BFMLALT \(by element\)](#): BFloat16 floating-point widening multiply-add long (by element).

[BFMLALB, BFMLALT \(vector\)](#): BFloat16 floating-point widening multiply-add long (vector).

[BFMMLA](#): BFloat16 floating-point matrix multiply-accumulate into 2x2 matrix.

[BIC \(vector, immediate\)](#): Bitwise bit Clear (vector, immediate).

[BIC \(vector, register\)](#): Bitwise bit Clear (vector, register).

[BIF](#): Bitwise Insert if False.

[BIT](#): Bitwise Insert if True.

[BSL](#): Bitwise Select.

[CLS \(vector\)](#): Count Leading Sign bits (vector).

[CLZ \(vector\)](#): Count Leading Zero bits (vector).

[CMEQ \(register\)](#): Compare bitwise Equal (vector).

[CMEQ \(zero\)](#): Compare bitwise Equal to zero (vector).

[CMGE \(register\)](#): Compare signed Greater than or Equal (vector).

[CMGE \(zero\)](#): Compare signed Greater than or Equal to zero (vector).

[CMGT \(register\)](#): Compare signed Greater than (vector).

[CMGT \(zero\)](#): Compare signed Greater than zero (vector).

[CMHI \(register\)](#): Compare unsigned Higher (vector).

[CMHS \(register\)](#): Compare unsigned Higher or Same (vector).

[CMLE \(zero\)](#): Compare signed Less than or Equal to zero (vector).

[CMLT \(zero\)](#): Compare signed Less than zero (vector).

[CMTST](#): Compare bitwise Test bits nonzero (vector).

[CNT](#): Population Count per byte.

[DUP \(element\)](#): Duplicate vector element to vector or scalar.

[DUP \(general\)](#): Duplicate general-purpose register to vector.

[EOR \(vector\)](#): Bitwise Exclusive-OR (vector).

[EOR3](#): Three-way Exclusive-OR.

[EXT](#): Extract vector from pair of vectors.

[FABD](#): Floating-point Absolute Difference (vector).

[FABS \(scalar\)](#): Floating-point Absolute value (scalar).

[FABS \(vector\)](#): Floating-point Absolute value (vector).

[FACGE](#): Floating-point Absolute Compare Greater than or Equal (vector).

[FACGT](#): Floating-point Absolute Compare Greater than (vector).

[FADD \(scalar\)](#): Floating-point Add (scalar).

[FADD \(vector\)](#): Floating-point Add (vector).

[FADDP \(scalar\)](#): Floating-point Add Pair of elements (scalar).

[FADDP \(vector\)](#): Floating-point Add Pairwise (vector).

[FCADD](#): Floating-point Complex Add.

[FCCMP](#): Floating-point Conditional quiet Compare (scalar).

[FCCMPE](#): Floating-point Conditional signaling Compare (scalar).

[FCMEQ \(register\)](#): Floating-point Compare Equal (vector).

[FCMEQ \(zero\)](#): Floating-point Compare Equal to zero (vector).

[FCMGE \(register\)](#): Floating-point Compare Greater than or Equal (vector).

[FCMGE \(zero\)](#): Floating-point Compare Greater than or Equal to zero (vector).

[FCMGT \(register\)](#): Floating-point Compare Greater than (vector).

[FCMGT \(zero\)](#): Floating-point Compare Greater than zero (vector).

[FCMLA](#): Floating-point Complex Multiply Accumulate.

[FCMLA \(by element\)](#): Floating-point Complex Multiply Accumulate (by element).

[FCMLE \(zero\)](#): Floating-point Compare Less than or Equal to zero (vector).

[FCMLT \(zero\)](#): Floating-point Compare Less than zero (vector).

[FCMP](#): Floating-point quiet Compare (scalar).

[FCMPE](#): Floating-point signaling Compare (scalar).

[FCSEL](#): Floating-point Conditional Select (scalar).

[FCVT](#): Floating-point Convert precision (scalar).

[FCVTAS \(scalar\)](#): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar).

[FCVTAS \(vector\)](#): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector).

[FCVTAU \(scalar\)](#): Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar).

[FCVTAU \(vector\)](#): Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector).

[FCVTL, FCVTL2](#): Floating-point Convert to higher precision Long (vector).

[FCVTMS \(scalar\)](#): Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar).

[FCVTMS \(vector\)](#): Floating-point Convert to Signed integer, rounding toward Minus infinity (vector).

[FCVTMU \(scalar\)](#): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar).

[FCVTMU \(vector\)](#): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector).

[FCVTN, FCVTN2](#): Floating-point Convert to lower precision Narrow (vector).

[FCVTNS \(scalar\)](#): Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar).

[FCVTNS \(vector\)](#): Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector).

[FCVTNU \(scalar\)](#): Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar).

[FCVTNU \(vector\)](#): Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector).

[FCVTPS \(scalar\)](#): Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar).

[FCVTPS \(vector\)](#): Floating-point Convert to Signed integer, rounding toward Plus infinity (vector).

[FCVTPU \(scalar\)](#): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar).

[FCVTPU \(vector\)](#): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector).

[FCVTXN, FCVTXN2](#): Floating-point Convert to lower precision Narrow, rounding to odd (vector).

[FCVTZS \(scalar, fixed-point\)](#): Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar).

[FCVTZS \(scalar, integer\)](#): Floating-point Convert to Signed integer, rounding toward Zero (scalar).

[FCVTZS \(vector, fixed-point\)](#): Floating-point Convert to Signed fixed-point, rounding toward Zero (vector).

[FCVTZS \(vector, integer\)](#): Floating-point Convert to Signed integer, rounding toward Zero (vector).

[FCVTZU \(scalar, fixed-point\)](#): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar).

[FCVTZU \(scalar, integer\)](#): Floating-point Convert to Unsigned integer, rounding toward Zero (scalar).

[FCVTZU \(vector, fixed-point\)](#): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector).

[FCVTZU \(vector, integer\)](#): Floating-point Convert to Unsigned integer, rounding toward Zero (vector).

[FDIV \(scalar\)](#): Floating-point Divide (scalar).

[FDIV \(vector\)](#): Floating-point Divide (vector).

[FJCVTZS](#): Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero.

[FMADD](#): Floating-point fused Multiply-Add (scalar).

[FMAX \(scalar\)](#): Floating-point Maximum (scalar).

[FMAX \(vector\)](#): Floating-point Maximum (vector).

[FMAXNM \(scalar\)](#): Floating-point Maximum Number (scalar).

[FMAXNM \(vector\)](#): Floating-point Maximum Number (vector).

[FMAXNMP \(scalar\)](#): Floating-point Maximum Number of Pair of elements (scalar).

[FMAXNMP \(vector\)](#): Floating-point Maximum Number Pairwise (vector).

[FMAXNMV](#): Floating-point Maximum Number across Vector.

[FMAXP \(scalar\)](#): Floating-point Maximum of Pair of elements (scalar).

[FMAXP \(vector\)](#): Floating-point Maximum Pairwise (vector).

[FMAXV](#): Floating-point Maximum across Vector.

[FMIN \(scalar\)](#): Floating-point Minimum (scalar).

[FMIN \(vector\)](#): Floating-point minimum (vector).

[FMINNM \(scalar\)](#): Floating-point Minimum Number (scalar).

[FMINNM \(vector\)](#): Floating-point Minimum Number (vector).

[FMINNMP \(scalar\)](#): Floating-point Minimum Number of Pair of elements (scalar).

[FMINNMP \(vector\)](#): Floating-point Minimum Number Pairwise (vector).

[FMINNMV](#): Floating-point Minimum Number across Vector.

[FMINP \(scalar\)](#): Floating-point Minimum of Pair of elements (scalar).

[FMINP \(vector\)](#): Floating-point Minimum Pairwise (vector).

[FMINV](#): Floating-point Minimum across Vector.

[FMLA \(by element\)](#): Floating-point fused Multiply-Add to accumulator (by element).

[FMLA \(vector\)](#): Floating-point fused Multiply-Add to accumulator (vector).

[FMLAL, FMLAL2 \(by element\)](#): Floating-point fused Multiply-Add Long to accumulator (by element).

[FMLAL, FMLAL2 \(vector\)](#): Floating-point fused Multiply-Add Long to accumulator (vector).

[FMLS \(by element\)](#): Floating-point fused Multiply-Subtract from accumulator (by element).

[FMLS \(vector\)](#): Floating-point fused Multiply-Subtract from accumulator (vector).

[FMLS, FMLSL2 \(by element\)](#): Floating-point fused Multiply-Subtract Long from accumulator (by element).

[FMLS, FMLSL2 \(vector\)](#): Floating-point fused Multiply-Subtract Long from accumulator (vector).

[FMOV \(general\)](#): Floating-point Move to or from general-purpose register without conversion.

[FMOV \(register\)](#): Floating-point Move register without conversion.

[FMOV \(scalar, immediate\)](#): Floating-point move immediate (scalar).

[FMOV \(vector, immediate\)](#): Floating-point move immediate (vector).

[FMSUB](#): Floating-point Fused Multiply-Subtract (scalar).

[FMUL \(by element\)](#): Floating-point Multiply (by element).

[FMUL \(scalar\)](#): Floating-point Multiply (scalar).

[FMUL \(vector\)](#): Floating-point Multiply (vector).

[FMULX](#): Floating-point Multiply extended.

[FMULX \(by element\)](#): Floating-point Multiply extended (by element).

[FNEG \(scalar\)](#): Floating-point Negate (scalar).

[FNEG \(vector\)](#): Floating-point Negate (vector).

[FNMADD](#): Floating-point Negated fused Multiply-Add (scalar).

[FNMSUB](#): Floating-point Negated fused Multiply-Subtract (scalar).



[FNMUL \(scalar\)](#): Floating-point Multiply-Negate (scalar).

[FRECPE](#): Floating-point Reciprocal Estimate.

[FRECPS](#): Floating-point Reciprocal Step.

[FRECPX](#): Floating-point Reciprocal exponent (scalar).

[FRINT32X \(scalar\)](#): Floating-point Round to 32-bit Integer, using current rounding mode (scalar).

[FRINT32X \(vector\)](#): Floating-point Round to 32-bit Integer, using current rounding mode (vector).

[FRINT32Z \(scalar\)](#): Floating-point Round to 32-bit Integer toward Zero (scalar).

[FRINT32Z \(vector\)](#): Floating-point Round to 32-bit Integer toward Zero (vector).

[FRINT64X \(scalar\)](#): Floating-point Round to 64-bit Integer, using current rounding mode (scalar).

[FRINT64X \(vector\)](#): Floating-point Round to 64-bit Integer, using current rounding mode (vector).

[FRINT64Z \(scalar\)](#): Floating-point Round to 64-bit Integer toward Zero (scalar).

[FRINT64Z \(vector\)](#): Floating-point Round to 64-bit Integer toward Zero (vector).

[FRINTA \(scalar\)](#): Floating-point Round to Integral, to nearest with ties to Away (scalar).

[FRINTA \(vector\)](#): Floating-point Round to Integral, to nearest with ties to Away (vector).

[FRINTI \(scalar\)](#): Floating-point Round to Integral, using current rounding mode (scalar).

[FRINTI \(vector\)](#): Floating-point Round to Integral, using current rounding mode (vector).

[FRINTM \(scalar\)](#): Floating-point Round to Integral, toward Minus infinity (scalar).

[FRINTM \(vector\)](#): Floating-point Round to Integral, toward Minus infinity (vector).

[FRINTN \(scalar\)](#): Floating-point Round to Integral, to nearest with ties to even (scalar).

[FRINTN \(vector\)](#): Floating-point Round to Integral, to nearest with ties to even (vector).

[FRINTP \(scalar\)](#): Floating-point Round to Integral, toward Plus infinity (scalar).

[FRINTP \(vector\)](#): Floating-point Round to Integral, toward Plus infinity (vector).

[FRINTX \(scalar\)](#): Floating-point Round to Integral exact, using current rounding mode (scalar).

[FRINTX \(vector\)](#): Floating-point Round to Integral exact, using current rounding mode (vector).

[FRINTZ \(scalar\)](#): Floating-point Round to Integral, toward Zero (scalar).

[FRINTZ \(vector\)](#): Floating-point Round to Integral, toward Zero (vector).

[FRSQRT](#): Floating-point Reciprocal Square Root Estimate.

[FRSQRTS](#): Floating-point Reciprocal Square Root Step.

[FSQRT \(scalar\)](#): Floating-point Square Root (scalar).

[FSQRT \(vector\)](#): Floating-point Square Root (vector).

[FSUB \(scalar\)](#): Floating-point Subtract (scalar).

[FSUB \(vector\)](#): Floating-point Subtract (vector).

[INS \(element\)](#): Insert vector element from another vector element.

[INS \(general\)](#): Insert vector element from general-purpose register.

[LD1 \(multiple structures\)](#): Load multiple single-element structures to one, two, three, or four registers.

[LD1 \(single structure\)](#): Load one single-element structure to one lane of one register.

[LD1R](#): Load one single-element structure and Replicate to all lanes (of one register).

[LD2 \(multiple structures\)](#): Load multiple 2-element structures to two registers.

[LD2 \(single structure\)](#): Load single 2-element structure to one lane of two registers.

[LD2R](#): Load single 2-element structure and Replicate to all lanes of two registers.

[LD3 \(multiple structures\)](#): Load multiple 3-element structures to three registers.

[LD3 \(single structure\)](#): Load single 3-element structure to one lane of three registers.

[LD3R](#): Load single 3-element structure and Replicate to all lanes of three registers.

[LD4 \(multiple structures\)](#): Load multiple 4-element structures to four registers.

[LD4 \(single structure\)](#): Load single 4-element structure to one lane of four registers.

[LD4R](#): Load single 4-element structure and Replicate to all lanes of four registers.

[LDAP1 \(SIMD&FP\)](#): Load-Acquire RCpc one single-element structure to one lane of one register.

[LDAPUR \(SIMD&FP\)](#): Load-Acquire RCpc SIMD&FP Register (unscaled offset).

[LDNP \(SIMD&FP\)](#): Load Pair of SIMD&FP registers, with Non-temporal hint.

[LDP \(SIMD&FP\)](#): Load Pair of SIMD&FP registers.

[LDR \(immediate, SIMD&FP\)](#): Load SIMD&FP Register (immediate offset).

[LDR \(literal, SIMD&FP\)](#): Load SIMD&FP Register (PC-relative literal).

[LDR \(register, SIMD&FP\)](#): Load SIMD&FP Register (register offset).

[LDUR \(SIMD&FP\)](#): Load SIMD&FP Register (unscaled offset).

[MLA \(by element\)](#): Multiply-Add to accumulator (vector, by element).

[MLA \(vector\)](#): Multiply-Add to accumulator (vector).

[MLS \(by element\)](#): Multiply-Subtract from accumulator (vector, by element).

[MLS \(vector\)](#): Multiply-Subtract from accumulator (vector).

[MOV \(element\)](#): Move vector element to another vector element: an alias of INS (element).

[MOV \(from general\)](#): Move general-purpose register to a vector element: an alias of INS (general).

[MOV \(scalar\)](#): Move vector element to scalar: an alias of DUP (element).

[MOV \(to general\)](#): Move vector element to general-purpose register: an alias of UMOV.

[MOV \(vector\)](#): Move vector: an alias of ORR (vector, register).

[MOVI](#): Move Immediate (vector).

[MUL \(by element\)](#): Multiply (vector, by element).

[MUL \(vector\)](#): Multiply (vector).

[MVN](#): Bitwise NOT (vector): an alias of NOT.

[MVNI](#): Move inverted Immediate (vector).

[NEG \(vector\)](#): Negate (vector).

[NOT](#): Bitwise NOT (vector).

[ORN \(vector\)](#): Bitwise inclusive OR NOT (vector).

[ORR \(vector, immediate\)](#): Bitwise inclusive OR (vector, immediate).

[ORR \(vector, register\)](#): Bitwise inclusive OR (vector, register).

[PMUL](#): Polynomial Multiply.

[PMULL, PMULL2](#): Polynomial Multiply Long.

[RADDHN, RADDHN2](#): Rounding Add returning High Narrow.

[RAX1](#): Rotate and Exclusive-OR.

[RBIT \(vector\)](#): Reverse Bit order (vector).

[REV16 \(vector\)](#): Reverse elements in 16-bit halfwords (vector).

[REV32 \(vector\)](#): Reverse elements in 32-bit words (vector).

[REV64](#): Reverse elements in 64-bit doublewords (vector).

[RSHRN, RSHRN2](#): Rounding Shift Right Narrow (immediate).

[RSUBHN, RSUBHN2](#): Rounding Subtract returning High Narrow.

[SABA](#): Signed Absolute difference and Accumulate.

[SABAL, SABAL2](#): Signed Absolute difference and Accumulate Long.

[SABD](#): Signed Absolute Difference.

[SABDL, SABDL2](#): Signed Absolute Difference Long.

[SADALP](#): Signed Add and Accumulate Long Pairwise.

[SADDL, SADDL2](#): Signed Add Long (vector).

[SADDLP](#): Signed Add Long Pairwise.

[SADDLV](#): Signed Add Long across Vector.

[SADDW, SADDW2](#): Signed Add Wide.

[SCVTF \(scalar, fixed-point\)](#): Signed fixed-point Convert to Floating-point (scalar).

[SCVTF \(scalar, integer\)](#): Signed integer Convert to Floating-point (scalar).

[SCVTF \(vector, fixed-point\)](#): Signed fixed-point Convert to Floating-point (vector).

[SCVTF \(vector, integer\)](#): Signed integer Convert to Floating-point (vector).

[SDOT \(by element\)](#): Dot Product signed arithmetic (vector, by element).

[SDOT \(vector\)](#): Dot Product signed arithmetic (vector).

[SHA1C](#): SHA1 hash update (choose).

[SHA1H](#): SHA1 fixed rotate.

[SHA1M](#): SHA1 hash update (majority).

[SHA1P](#): SHA1 hash update (parity).

[SHA1SU0](#): SHA1 schedule update 0.

[SHA1SU1](#): SHA1 schedule update 1.

[SHA256H](#): SHA256 hash update (part 1).

[SHA256H2](#): SHA256 hash update (part 2).

[SHA256SU0](#): SHA256 schedule update 0.

[SHA256SU1](#): SHA256 schedule update 1.

[SHA512H](#): SHA512 Hash update part 1.

[SHA512H2](#): SHA512 Hash update part 2.

[SHA512SU0](#): SHA512 Schedule Update 0.

[SHA512SU1](#): SHA512 Schedule Update 1.

[SHADD](#): Signed Halving Add.

[SHL](#): Shift Left (immediate).

[SHLL](#), [SHLL2](#): Shift Left Long (by element size).

[SHRN](#), [SHRN2](#): Shift Right Narrow (immediate).

[SHSUB](#): Signed Halving Subtract.

[SLI](#): Shift Left and Insert (immediate).

[SM3PARTW1](#): SM3PARTW1.

[SM3PARTW2](#): SM3PARTW2.

[SM3SS1](#): SM3SS1.

[SM3TT1A](#): SM3TT1A.

[SM3TT1B](#): SM3TT1B.

[SM3TT2A](#): SM3TT2A.

[SM3TT2B](#): SM3TT2B.

[SM4E](#): SM4 Encode.

[SM4EKEY](#): SM4 Key.

[SMAX](#): Signed Maximum (vector).

[SMAXP](#): Signed Maximum Pairwise.

[SMAXV](#): Signed Maximum across Vector.

[SMIN](#): Signed Minimum (vector).

[SMINP](#): Signed Minimum Pairwise.

[SMINV](#): Signed Minimum across Vector.

[SMLAL](#), [SMLAL2 \(by element\)](#): Signed Multiply-Add Long (vector, by element).

[SMLAL](#), [SMLAL2 \(vector\)](#): Signed Multiply-Add Long (vector).

[SMLSL](#), [SMLSL2 \(by element\)](#): Signed Multiply-Subtract Long (vector, by element).

[SMLSL](#), [SMLSL2 \(vector\)](#): Signed Multiply-Subtract Long (vector).

[SMMLA \(vector\)](#): Signed 8-bit integer matrix multiply-accumulate (vector).

[SMOV](#): Signed Move vector element to general-purpose register.

[SMULL](#), [SMULL2 \(by element\)](#): Signed Multiply Long (vector, by element).

[SMULL](#), [SMULL2 \(vector\)](#): Signed Multiply Long (vector).

[SQABS](#): Signed saturating Absolute value.

[SQADD](#): Signed saturating Add.

[SQDMLAL](#), [SQDMLAL2 \(by element\)](#): Signed saturating Doubling Multiply-Add Long (by element).

[SQDMLAL, SQDMLAL2 \(vector\)](#): Signed saturating Doubling Multiply-Add Long.

[SQDMLSL, SQDMLSL2 \(by element\)](#): Signed saturating Doubling Multiply-Subtract Long (by element).

[SQDMLSL, SQDMLSL2 \(vector\)](#): Signed saturating Doubling Multiply-Subtract Long.

[SQDMULH \(by element\)](#): Signed saturating Doubling Multiply returning High half (by element).

[SQDMULH \(vector\)](#): Signed saturating Doubling Multiply returning High half.

[SQDMULL, SQDMULL2 \(by element\)](#): Signed saturating Doubling Multiply Long (by element).

[SQDMULL, SQDMULL2 \(vector\)](#): Signed saturating Doubling Multiply Long.

[SQNEG](#): Signed saturating Negate.

[SQRDMLAH \(by element\)](#): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element).

[SQRDMLAH \(vector\)](#): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector).

[SQRDMLSH \(by element\)](#): Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element).

[SQRDMLSH \(vector\)](#): Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector).

[SQRDMULH \(by element\)](#): Signed saturating Rounding Doubling Multiply returning High half (by element).

[SQRDMULH \(vector\)](#): Signed saturating Rounding Doubling Multiply returning High half.

[SQRSHL](#): Signed saturating Rounding Shift Left (register).

[SQRSHRN, SQRSHRN2](#): Signed saturating Rounded Shift Right Narrow (immediate).

[SQRSHRUN, SQRSHRUN2](#): Signed saturating Rounded Shift Right Unsigned Narrow (immediate).

[SQSHL \(immediate\)](#): Signed saturating Shift Left (immediate).

[SQSHL \(register\)](#): Signed saturating Shift Left (register).

[SQSHLU](#): Signed saturating Shift Left Unsigned (immediate).

[SQSHRN, SQSHRN2](#): Signed saturating Shift Right Narrow (immediate).

[SQSHRUN, SQSHRUN2](#): Signed saturating Shift Right Unsigned Narrow (immediate).

[SQSUB](#): Signed saturating Subtract.

[SQXTN, SQXTN2](#): Signed saturating extract Narrow.

[SQXTUN, SQXTUN2](#): Signed saturating extract Unsigned Narrow.

[SRHADD](#): Signed Rounding Halving Add.

[SRI](#): Shift Right and Insert (immediate).

[SRSHL](#): Signed Rounding Shift Left (register).

[SRSHR](#): Signed Rounding Shift Right (immediate).

[SRSRA](#): Signed Rounding Shift Right and Accumulate (immediate).

[SSHL](#): Signed Shift Left (register).

[SSHLL, SSHLL2](#): Signed Shift Left Long (immediate).

[SSHR](#): Signed Shift Right (immediate).

[SSRA](#): Signed Shift Right and Accumulate (immediate).

[SSUBL, SSUBL2](#): Signed Subtract Long.

[SSUBW, SSUBW2](#): Signed Subtract Wide.

[ST1 \(multiple structures\)](#): Store multiple single-element structures from one, two, three, or four registers.

[ST1 \(single structure\)](#): Store a single-element structure from one lane of one register.

[ST2 \(multiple structures\)](#): Store multiple 2-element structures from two registers.

[ST2 \(single structure\)](#): Store single 2-element structure from one lane of two registers.

[ST3 \(multiple structures\)](#): Store multiple 3-element structures from three registers.

[ST3 \(single structure\)](#): Store single 3-element structure from one lane of three registers.

[ST4 \(multiple structures\)](#): Store multiple 4-element structures from four registers.

[ST4 \(single structure\)](#): Store single 4-element structure from one lane of four registers.

[STL1 \(SIMD&FP\)](#): Store-Release a single-element structure from one lane of one register.

[STLUR \(SIMD&FP\)](#): Store-Release SIMD&FP Register (unscaled offset).

[STNP \(SIMD&FP\)](#): Store Pair of SIMD&FP registers, with Non-temporal hint.

[STP \(SIMD&FP\)](#): Store Pair of SIMD&FP registers.

[STR \(immediate, SIMD&FP\)](#): Store SIMD&FP register (immediate offset).

[STR \(register, SIMD&FP\)](#): Store SIMD&FP register (register offset).

[STUR \(SIMD&FP\)](#): Store SIMD&FP register (unscaled offset).

[SUB \(vector\)](#): Subtract (vector).

[SUBHN, SUBHN2](#): Subtract returning High Narrow.

[SUDOT \(by element\)](#): Dot product with signed and unsigned integers (vector, by element).

[SUQADD](#): Signed saturating Accumulate of Unsigned value.

[SXTL, SXTL2](#): Signed extend Long: an alias of SSHLL, SSHLL2.

[TBL](#): Table vector Lookup.

[TBX](#): Table vector lookup extension.

[TRN1](#): Transpose vectors (primary).

[TRN2](#): Transpose vectors (secondary).

[UABA](#): Unsigned Absolute difference and Accumulate.

[UABAL, UABAL2](#): Unsigned Absolute difference and Accumulate Long.

[UABD](#): Unsigned Absolute Difference (vector).

[UABDL, UABDL2](#): Unsigned Absolute Difference Long.

[UADALP](#): Unsigned Add and Accumulate Long Pairwise.

[UADDL, UADDL2](#): Unsigned Add Long (vector).

[UADDLP](#): Unsigned Add Long Pairwise.

[UADDLV](#): Unsigned sum Long across Vector.

[UADDW, UADDW2](#): Unsigned Add Wide.

[UCVTF \(scalar, fixed-point\)](#): Unsigned fixed-point Convert to Floating-point (scalar).

[UCVTF \(scalar, integer\)](#): Unsigned integer Convert to Floating-point (scalar).

[UCVTF \(vector, fixed-point\)](#): Unsigned fixed-point Convert to Floating-point (vector).

[UCVTF \(vector, integer\)](#): Unsigned integer Convert to Floating-point (vector).

[UDOT \(by element\)](#): Dot Product unsigned arithmetic (vector, by element).

[UDOT \(vector\)](#): Dot Product unsigned arithmetic (vector).

[UHADD](#): Unsigned Halving Add.

[UHSUB](#): Unsigned Halving Subtract.

[UMAX](#): Unsigned Maximum (vector).

[UMAXP](#): Unsigned Maximum Pairwise.

[UMAXV](#): Unsigned Maximum across Vector.

[UMIN](#): Unsigned Minimum (vector).

[UMINP](#): Unsigned Minimum Pairwise.

[UMINV](#): Unsigned Minimum across Vector.

[UMLAL, UMLAL2 \(by element\)](#): Unsigned Multiply-Add Long (vector, by element).

[UMLAL, UMLAL2 \(vector\)](#): Unsigned Multiply-Add Long (vector).

[UMLSL, UMLSL2 \(by element\)](#): Unsigned Multiply-Subtract Long (vector, by element).

[UMLSL, UMLSL2 \(vector\)](#): Unsigned Multiply-Subtract Long (vector).

[UMMLA \(vector\)](#): Unsigned 8-bit integer matrix multiply-accumulate (vector).

[UMOV](#): Unsigned Move vector element to general-purpose register.

[UMULL, UMULL2 \(by element\)](#): Unsigned Multiply Long (vector, by element).

[UMULL, UMULL2 \(vector\)](#): Unsigned Multiply long (vector).

[UQADD](#): Unsigned saturating Add.

[UQRSHL](#): Unsigned saturating Rounding Shift Left (register).

[UQRSHRN, UQRSHRN2](#): Unsigned saturating Rounded Shift Right Narrow (immediate).

[UQSHL \(immediate\)](#): Unsigned saturating Shift Left (immediate).

[UQSHL \(register\)](#): Unsigned saturating Shift Left (register).

[UQSHRN, UQSHRN2](#): Unsigned saturating Shift Right Narrow (immediate).

[UQSUB](#): Unsigned saturating Subtract.

[UQXTN, UQXTN2](#): Unsigned saturating extract Narrow.

[URECPE](#): Unsigned Reciprocal Estimate.

[URHADD](#): Unsigned Rounding Halving Add.

[URSHL](#): Unsigned Rounding Shift Left (register).

[URSHR](#): Unsigned Rounding Shift Right (immediate).

[URSQRTE](#): Unsigned Reciprocal Square Root Estimate.

[URSRA](#): Unsigned Rounding Shift Right and Accumulate (immediate).

[USDOT \(by element\)](#): Dot Product with unsigned and signed integers (vector, by element).

[USDOT \(vector\)](#): Dot Product with unsigned and signed integers (vector).

[USHL](#): Unsigned Shift Left (register).

[USHLL, USHLL2](#): Unsigned Shift Left Long (immediate).

[USHR](#): Unsigned Shift Right (immediate).

[USMMLA \(vector\)](#): Unsigned and signed 8-bit integer matrix multiply-accumulate (vector).

[USQADD](#): Unsigned saturating Accumulate of Signed value.

[USRA](#): Unsigned Shift Right and Accumulate (immediate).

[USUBL, USUBL2](#): Unsigned Subtract Long.

[USUBW, USUBW2](#): Unsigned Subtract Wide.

[UXTL, UXTL2](#): Unsigned extend Long: an alias of USHLL, USHLL2.

[UZP1](#): Unzip vectors (primary).

[UZP2](#): Unzip vectors (secondary).

[XAR](#): Exclusive-OR and Rotate.

[XTN, XTN2](#): Extract Narrow.

[ZIP1](#): Zip vectors (primary).

[ZIP2](#): Zip vectors (secondary).

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



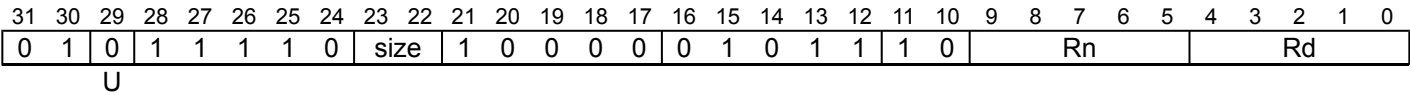
ABS

Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD&FP register, puts the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar

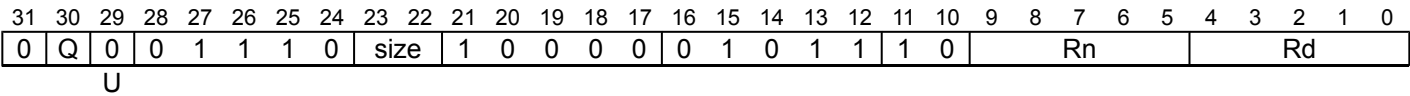


ABS <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector



ABS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

- <V> Is a width specifier, encoded in “size”:
- | size | <V>      |
|------|----------|
| 0x   | RESERVED |
| 10   | RESERVED |
| 11   | D        |
- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

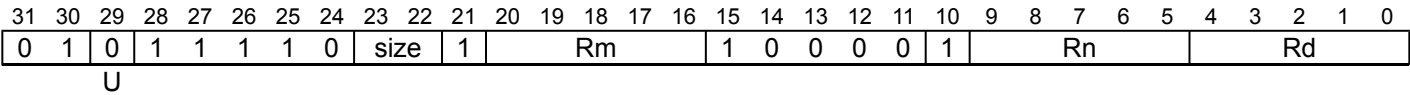
ADD (vector)

Add (vector). This instruction adds corresponding elements in the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

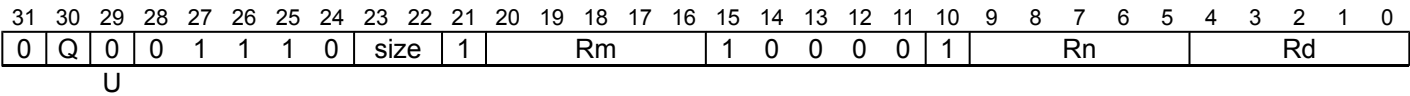
Scalar



ADD <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');
```

Vector



ADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;

V[d, datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

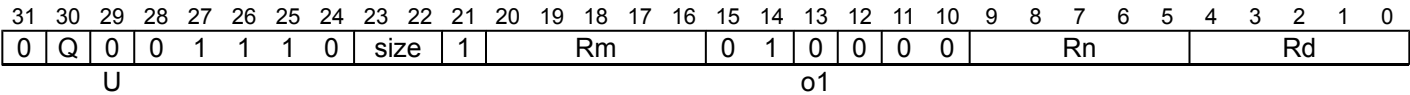
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADDHN, ADDHN2

Add returning High Narrow. This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The results are truncated. For rounded results, see [RADDHN](#). The ADDHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the ADDHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register. Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



ADDHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(2*datasize) operand2 = V[m, 2*datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, 2*esize]);
    element2 = UInt(Elem[operand2, e, 2*esize]);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = RShr(sum, esize, round);
    Elem[result, e, esize] = sum<esize-1:0>;

Vpart[d, part, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADDP (scalar)

Add Pair of elements (scalar). This instruction adds two vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		1	1	0	0	0	1	1	0	1	1	1	0	Rn				Rd					

ADDP <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;

integer esize = 8 << UInt(size);
integer datasize = esize * 2;
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is the source arrangement specifier, encoded in “size”:

size	<T>
0x	RESERVED
10	RESERVED
11	2D

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_ADD, operand, esize);
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

ADDP (vector)

Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	Rm						1	0	1	1	1	1	Rn						Rd			

```
ADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
    Elem[result, e, esize] = element1 + element2;

V[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.



- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADDV

Add across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	1	0	0	0	1	1	0	1	1	1	0	1	0	Rn				Rd				

ADDV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_ADD, operand, esize);
```

Operational information

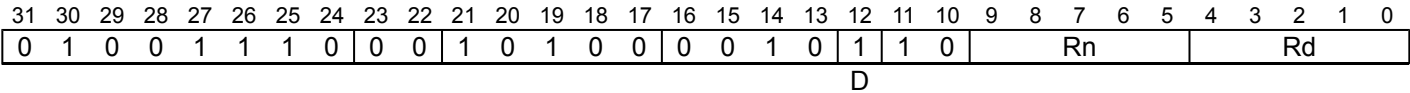
- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



AESD

AES single round decryption.

Advanced SIMD  
(FEAT\_AES)



AESD <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESEExt() then UNDEFINED;
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
bits(128) operand1 = V[d, 128];
bits(128) operand2 = V[n, 128];
bits(128) result;
result = operand1 EOR operand2;
result = AESInvSubBytes(AESInvShiftRows(result));
V[d, 128] = result;
```

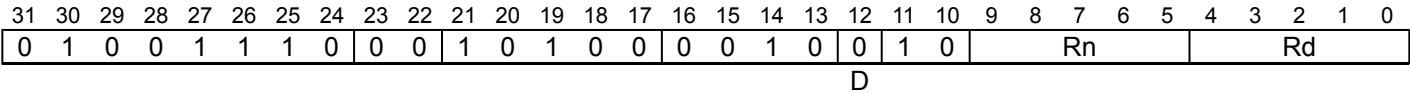
Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

AESE

AES single round encryption.

Advanced SIMD  
(FEAT\_AES)



AESE <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d, 128];
bits(128) operand2 = V[n, 128];
bits(128) result;
result = operand1 EOR operand2;
result = AESSubBytes(AESShiftRows(result));

V[d, 128] = result;
```

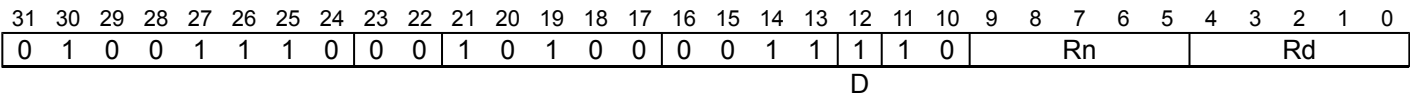
Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

AESIMC

AES inverse mix columns.

Advanced SIMD  
(FEAT\_AES)



AESIMC <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAEEExt() then UNDEFINED;
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
bits(128) operand = V[n, 128];
bits(128) result;
result = AESInvMixColumns(operand);
V[d, 128] = result;
```

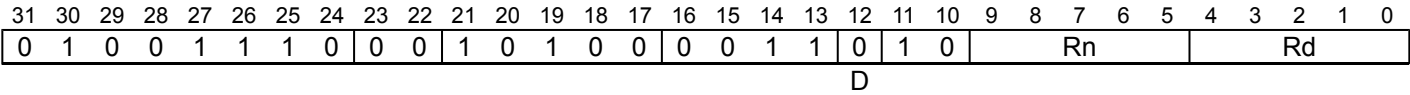
Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

AESMC

AES mix columns.

Advanced SIMD  
(FEAT\_AES)



AESMC <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESEExt() then UNDEFINED;
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand = V[n, 128];
bits(128) result;
result = AESMixColumns(operand);
V[d, 128] = result;
```

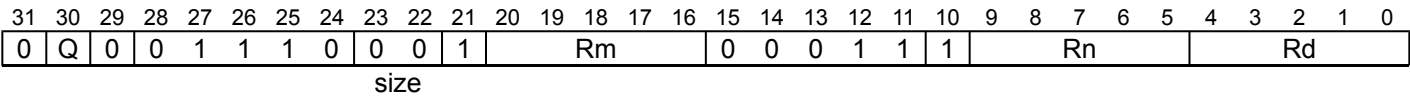
Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

AND (vector)

Bitwise AND (vector). This instruction performs a bitwise AND between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



AND <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "Q":
- | Q | <T> |
|---|-----|
| 0 | 8B  |
| 1 | 16B |
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

result = operand1 AND operand2;
V[d, datasize] = result;
```

Operational information

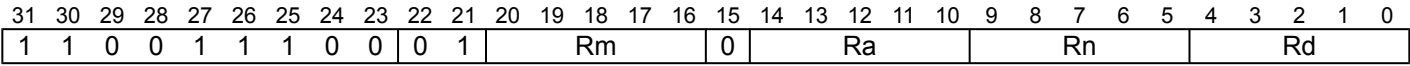
- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



BCAX

Bit Clear and exclusive-OR performs a bitwise AND of the 128-bit vector in a source SIMD&FP register and the complement of the vector in another source SIMD&FP register, then performs a bitwise exclusive-OR of the resulting vector and the vector in a third source SIMD&FP register, and writes the result to the destination SIMD&FP register. This instruction is implemented only when *FEAT\_SHA3* is implemented.

Advanced SIMD  
(FEAT\_SHA3)



BCAX <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B

```
if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Va> Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Va = V[a, 128];
V[d, 128] = Vn EOR (Vm AND NOT(Va));
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

BFCVT

Floating-point convert from single-precision to BFloat16 format (scalar) converts the single-precision floating-point value in the 32-bit SIMD&FP source register to BFloat16 format and writes the result in the 16-bit SIMD&FP destination register.

[ID\\_AA64ISAR1\\_EL1](#).BF16 indicates whether this instruction is supported.

Single-precision to BFloat16  
(FEAT\_BF16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	0	1	1	0	0	0	1	1	0	1	0	0	0	0	Rn				Rd					

BFCVT <Hd>, <Sn>

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

bits(32) operand = V[n, 32];
FPCRTyp fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

Elem[result, 0, 16] = FPConvertBF(operand, fpcr);

V[d, 128] = result;
```

BFCVTN, BFCVTN2

Floating-point convert from single-precision to BFloat16 format (vector) reads each single-precision element in the SIMD&FP source vector, converts each value to BFloat16 format, and writes the results in the lower or upper half of the SIMD&FP destination vector. The result elements are half the width of the source elements.

The BFCVTN instruction writes the half-width results to the lower half of the destination vector and clears the upper half to zero, while the BFCVTN2 instruction writes the results to the upper half of the destination vector without affecting the other bits in the register.

Vector single-precision to BFloat16  
(FEAT\_BF16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	Q	0	0	1	1	1	0	1	0	1	0	0	0	0	1	0	1	1	0	1	0	Rn						Rd					

```
BFCVTN{2} <Vd>.<Ta>, <Vn>.4S
```

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer part = UInt(Q);
integer elements = 64 DIV 16;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	4H
1	8H

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) operand = V[n, 128];
bits(64) result;

for e = 0 to elements-1
    Elem[result, e, 16] = FPConvertBF(Elem[operand, e, 32], FPCR[]);
Vpart[d, part, 64] = result;
```

BFDOT (by element)

BFloat16 floating-point dot product (vector, by element). This instruction delimits the source vectors into pairs of BFloat16 elements. The BFloat16 pair within the second source vector is specified using an immediate index. The index range is from 0 to 3 inclusive.

If FEAT\_EBF16 is not implemented or *FPCR*.EBF is 0, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second source vector. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Flushes denormalized inputs and results to zero, as if *FPCR*.{FZ, FIZ} is {1, 1}.
- Disables alternative floating point behaviors, as if *FPCR*.AH is 0.

If FEAT\_EBF16 is implemented and *FPCR*.EBF is 1, then this instruction:

- Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second source vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as governed by *FPCR*.RMode, *FPCR*.FZ, *FPCR*.AH, and *FPCR*.FIZ.

Irrespective of FEAT\_EBF16 and *FPCR*.EBF, this instruction:

- Does not modify the cumulative *FPSR* exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the *FPCR* trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Generates only the default NaN, as if *FPCR*.DN is 1.

*ID\_AA64ISAR1\_EL1*.BF16 indicates whether this instruction is supported.

Vector  
(FEAT\_BF16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	1	L	M	Rm				1	1	1	1	H	0	Rn				Rd					

BFDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.2H[<index>]

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "Q":

Q	<Tb>
0	4H
1	8H

- <Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
- <index> Is the immediate index of a pair of 16-bit elements in the range 0 to 3, encoded in the "H:L" fields.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(128) operand2 = V[m, 128];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;

for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2*e+0, 16];
    bits(16) elt1_b = Elem[operand1, 2*e+1, 16];
    bits(16) elt2_a = Elem[operand2, 2*i+0, 16];
    bits(16) elt2_b = Elem[operand2, 2*i+1, 16];

    bits(32) sum = Elem[operand3, e, 32];
    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;

V[d, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFDOT (vector)

BFloat16 floating-point dot product (vector). This instruction delimits the source vectors into pairs of BFloat16 elements.

If FEAT\_EBF16 is not implemented or *FPCR*.EBF is 0, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Flushes denormalized inputs and results to zero, as if *FPCR*.{FZ, FIZ} is {1, 1}.
- Disables alternative floating point behaviors, as if *FPCR*.AH is 0.

If FEAT\_EBF16 is implemented and *FPCR*.EBF is 1, then this instruction:

- Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as governed by *FPCR*.RMode, *FPCR*.FZ, *FPCR*.AH, and *FPCR*.FIZ.

Irrespective of FEAT\_EBF16 and *FPCR*.EBF, this instruction:

- Does not modify the cumulative *FPSR* exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the *FPCR* trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Generates only the default NaN, as if *FPCR*.DN is 1.

*ID\_AA64ISAR1\_EL1*.BF16 indicates whether this instruction is supported.

Vector  
(FEAT\_BF16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	0	Rm				1	1	1	1	1	1	1	Rn				Rd					

BFDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

Assembler Symbols

- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S
- <Vn>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	4H
1	8H
- <Vm>

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;

for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2*e+0, 16];
    bits(16) elt1_b = Elem[operand1, 2*e+1, 16];
    bits(16) elt2_a = Elem[operand2, 2*e+0, 16];
    bits(16) elt2_b = Elem[operand2, 2*e+1, 16];

    bits(32) sum = Elem[operand3, e, 32];
    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;

V[d, datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFMLALB, BFMLALT (by element)

BFloat16 floating-point widening multiply-add long (by element) widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first source vector, and the indexed element in the second source vector from Bfloat16 to single-precision format. The instruction then multiplies and adds these values without intermediate rounding to single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector.

[ID\\_AA64ISAR1\\_EL1](#).BF16 indicates whether this instruction is supported.

Vector  
(FEAT\_BF16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	1	1	L	M	Rm				1	1	1	1	H	0	Rn				Rd					

BFMLAL<bt> <Vd>.4S, <Vn>.8H, <Vm>.H[<index>]

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt('0':Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer elements = 128 DIV 32;
integer sel = UInt(Q);
```

Assembler Symbols

<bt> Is the bottom or top element specifier, encoded in “Q”:

Q	<bt>
0	B
1	T

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.
- <index> Is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) result;
bits(128) operand1 = V[n, 128];
bits(128) operand2 = V[m, 128];
bits(128) operand3 = V[d, 128];
bits(16) element2 = Elem[operand2, index, 16];

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, 2*e+sel, 16];
    bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAddH(addend, element1, element2, FPCR[]);

V[d, 128] = result;
```

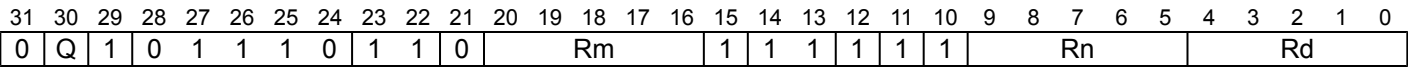


BFMLALB, BFMLALT (vector)

BFloat16 floating-point widening multiply-add long (vector) widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first and second source vectors from Bfloat16 to single-precision format. The instruction then multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors.

[ID\\_AA64ISAR1\\_EL1](#).BF16 indicates whether this instruction is supported.

Vector  
(FEAT\_BF16)



```
BFMLAL<bt> <Vd>.4S, <Vn>.8H, <Vm>.8H
```

```
if !HaveBF16Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer elements = 128 DIV 32;
integer sel = UInt(Q);
```

Assembler Symbols

- <bt> Is the bottom or top element specifier, encoded in “Q”:

Q	<bt>
0	B
1	T
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n, 128];
bits(128) operand2 = V[m, 128];
bits(128) operand3 = V[d, 128];
bits(128) result;

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, 2*e+sel, 16];
    bits(16) element2 = Elem[operand2, 2*e+sel, 16];
    bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAddH(addend, element1, element2, FPCR[]);
V[d, 128] = result;
```

BFMMLA

BFloat16 floating-point matrix multiply-accumulate into 2x2 matrix.

If FEAT\_EBF16 is not implemented or FPCR.EBF is 0, this instruction:

- Performs two unfused sums-of-products within each two pairs of adjacent BFloat16 elements while multiplying the 2x4 matrix of BFloat16 values in the first source vector with the 4x2 matrix of BFloat16 values in the second source vector. The intermediate single-precision products are rounded before they are summed and the intermediate sum is rounded before accumulation into the 2x2 single-precision matrix in the destination vector. This is equivalent to accumulating two 2-way unfused dot products per destination element.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Disables alternative floating point behaviors, as if FPCR.AH is 0.

If FEAT\_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

- Performs two fused sums-of-products within each two pairs of adjacent BFloat16 elements while multiplying the 2x4 matrix of BFloat16 values in the first source vector with the 4x2 matrix of BFloat16 values in the second source vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the 2x2 single-precision matrix in the destination vector. This is equivalent to accumulating two 2-way fused dot products per destination element.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as governed by FPCR.RMode, FPCR.FZ, FPCR.AH, and FPCR.FIZ.

Irrespective of FEAT\_EBF16 and FPCR.EBF, this instruction:

- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Generates only the default NaN, as if FPCR.DN is 1.

ID\_AA64ISAR1\_EL1.BF16 indicates whether this instruction is supported.

Vector  
(FEAT\_BF16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	0	0	1	0	Rm				1	1	1	0	1	1	Rn				Rd						

BFMMLA <Vd>.4S, <Vn>.8H, <Vm>.8H

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) op1 = V[n, 128];
bits(128) op2 = V[m, 128];
bits(128) acc = V[d, 128];

V[d, 128] = BFMatMulAdd(acc, op1, op2);
```

## Operational information

Arm expects that the BFMMLA instruction will deliver a peak BFloat16 multiply throughput that is at least as high as can be achieved using two BFDOT (vector) instructions, with a goal that it should have significantly higher throughput.

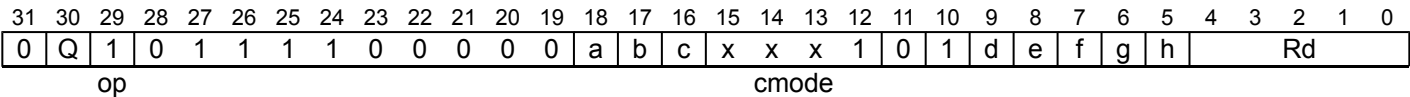
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BIC (vector, immediate)

Bitwise bit Clear (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise AND between each result and the complement of an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



16-bit (cmode == 10x1)

```
BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit (cmode == 0xx1)

```
BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
  when '0xx01' operation = ImmediateOp MVNI;
  when '0xx11' operation = ImmediateOp BIC;
  when '10x01' operation = ImmediateOp MVNI;
  when '10x11' operation = ImmediateOp BIC;
  when '110x1' operation = ImmediateOp MVNI;
  when '1110x' operation = ImmediateOp MOVI;
  when '11111'
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;
    operation = ImmediateOp MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP register, encoded in the "Rd" field.

<T> For the 16-bit variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit variant: is the shift amount encoded in "cmode<1>":

<b>cmode&lt;1&gt;</b>	<b>&lt;amount&gt;</b>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in “cmode<2:1>”:

<b>cmode&lt;2:1&gt;</b>	<b>&lt;amount&gt;</b>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
  when ImmediateOp_MOVI
    result = imm;
  when ImmediateOp_MVNI
    result = NOT(imm);
  when ImmediateOp_ORR
    operand = V[rd, datasize];
    result = operand OR imm;
  when ImmediateOp_BIC
    operand = V[rd, datasize];
    result = operand AND NOT(imm);

V[rd, datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

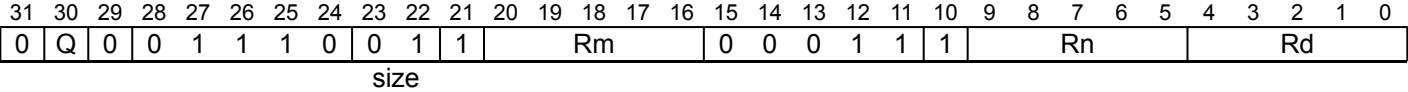
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BIC (vector, register)

Bitwise bit Clear (vector, register). This instruction performs a bitwise AND between the first source SIMD&FP register and the complement of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



```
BIC <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

operand2 = NOT(operand2);

result = operand1 AND operand2;
V[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

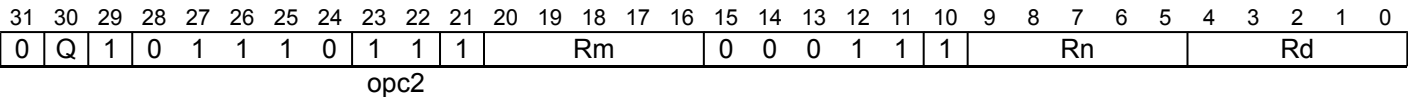
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BIF

Bitwise Insert if False. This instruction inserts each bit from the first source SIMD&FP register into the destination SIMD&FP register if the corresponding bit of the second source SIMD&FP register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



BIF <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand3;
bits(datasize) operand4 = V[n, datasize];

operand1 = V[d, datasize];
operand3 = NOT(V[m, datasize]);

V[d, datasize] = operand1 EOR ((operand1 EOR operand4) AND operand3);
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

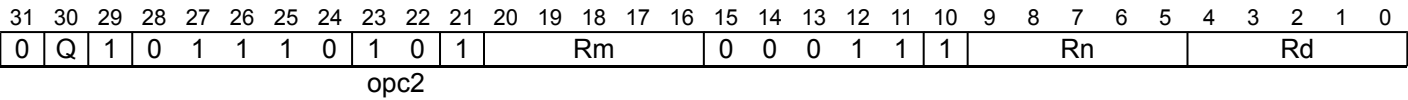
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BIT

Bitwise Insert if True. This instruction inserts each bit from the first source SIMD&FP register into the SIMD&FP destination register if the corresponding bit of the second source SIMD&FP register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



BIT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand3;
bits(datasize) operand4 = V[n, datasize];

operand1 = V[d, datasize];
operand3 = V[m, datasize];
V[d, datasize] = operand1 EOR ((operand1 EOR operand4) AND operand3);
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

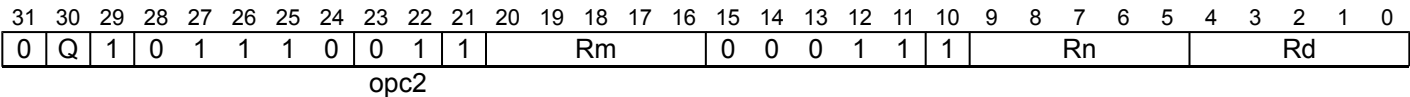
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



BSL

Bitwise Select. This instruction sets each bit in the destination SIMD&FP register to the corresponding bit from the first source SIMD&FP register when the original destination bit was 1, otherwise from the second source SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



BSL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand3;
bits(datasize) operand4 = V[n, datasize];

operand1 = V[m, datasize];
operand3 = V[d, datasize];
V[d, datasize] = operand1 EOR ((operand1 EOR operand4) AND operand3);
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

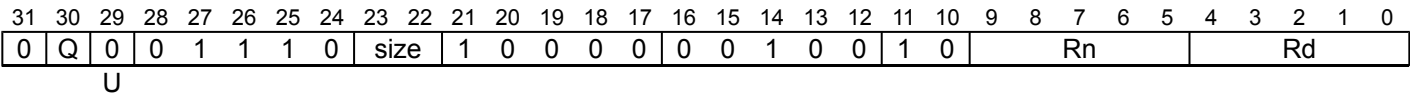
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CLS (vector)

Count Leading Sign bits (vector). This instruction counts the number of consecutive bits following the most significant bit that are the same as the most significant bit in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The count does not include the most significant bit itself.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



CLS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CountOp countop = if U == '1' then CountOp_CLZ else CountOp_CLS;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

integer count;
for e = 0 to elements-1
    if countop == CountOp_CLS then
        count = CountLeadingSignBits(Elem[operand, e, esize]);
    else
        count = CountLeadingZeroBits(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CLZ (vector)

Count Leading Zero bits (vector). This instruction counts the number of consecutive zeros, starting from the most significant bit, in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	0	0	0	1	0	0	1	0	Rn					Rd				
U																															

CLZ <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CountOp countop = if U == '1' then CountOp_CLZ else CountOp_CLS;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

integer count;
for e = 0 to elements-1
    if countop == CountOp_CLS then
        count = CountLeadingSignBits(Elem[operand, e, esize]);
    else
        count = CountLeadingZeroBits(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

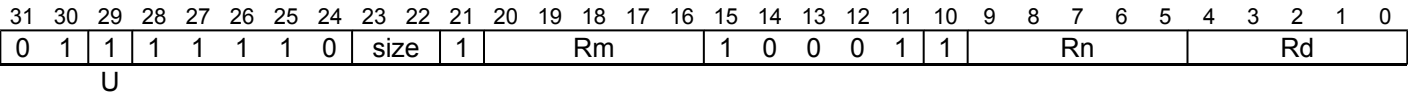
CMEQ (register)

Compare bitwise Equal (vector). This instruction compares each vector element from the first source SIMD&FP register with the corresponding vector element from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

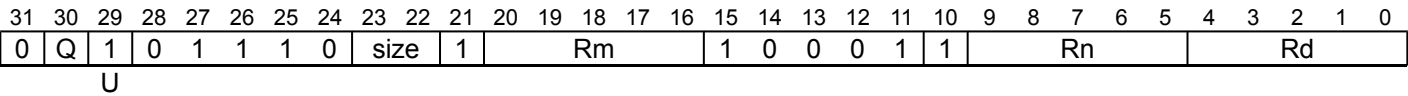
Scalar



CMEQ <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');
```

Vector



CMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if and_test then
        test_passed = !IsZero(element1 AND element2);
    else
        test_passed = (element1 == element2);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

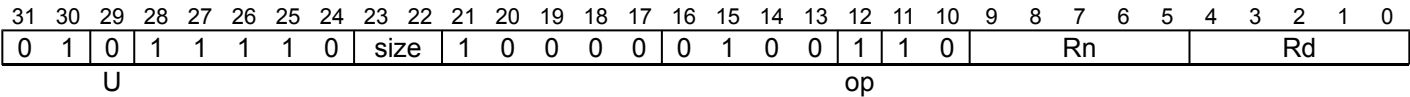
## CMEQ (zero)

Compare bitwise Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar



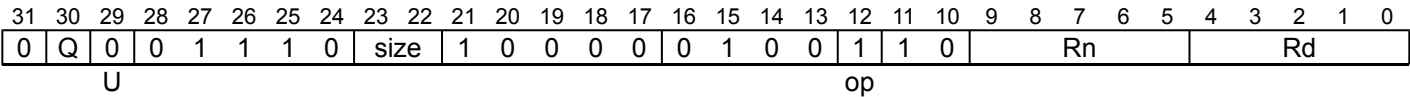
CMEQ <V><d>, <V><n>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Vector



CMEQ <Vd>.<T>, <Vn>.<T>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Assembler Symbols

<V> Is a width specifier, encoded in “size”:



size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

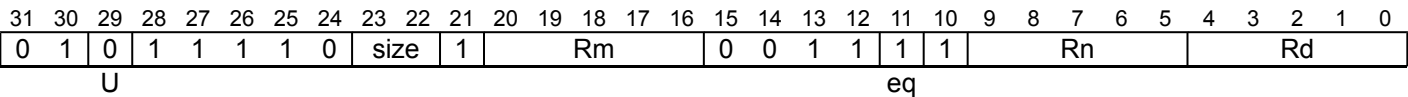
CMGE (register)

Compare signed Greater than or Equal (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than or equal to the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

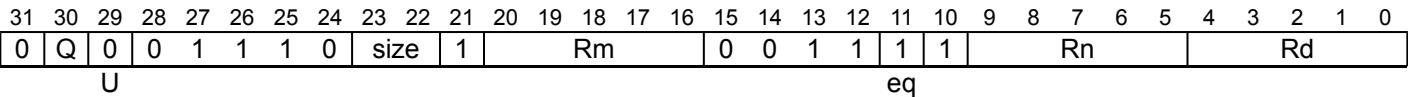
Scalar



CMGE <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Vector



CMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

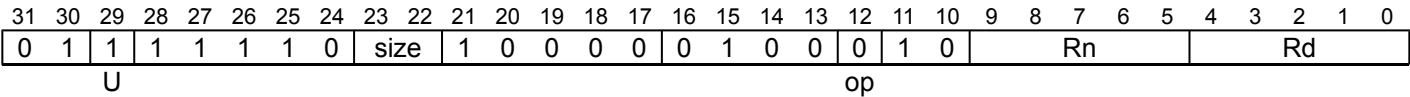
## CMGE (zero)

Compare signed Greater than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar



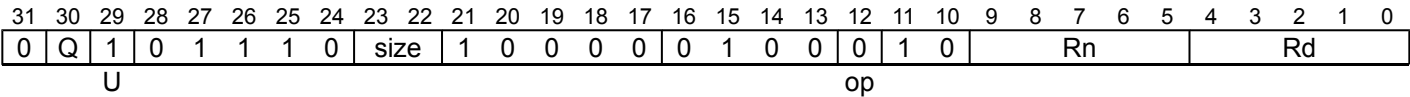
CMGE <V><d>, <V><n>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Vector



CMGE <Vd>.<T>, <Vn>.<T>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

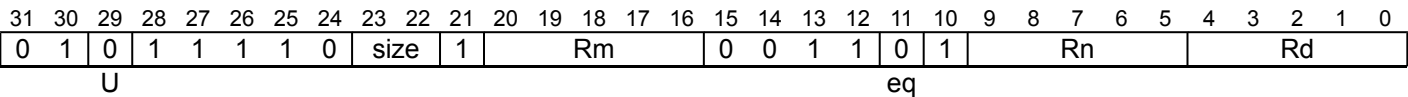
CMGT (register)

Compare signed Greater than (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

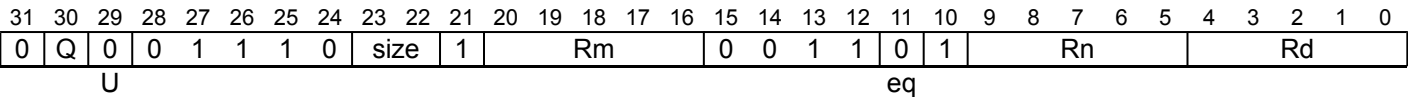
Scalar



CMGT <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Vector



CMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

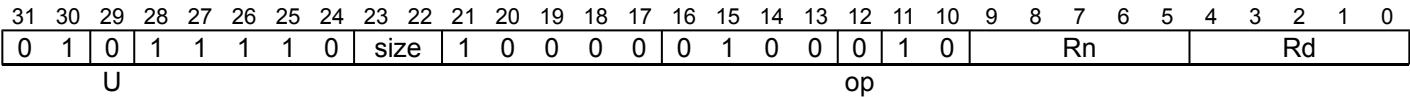
## CMGT (zero)

Compare signed Greater than zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar



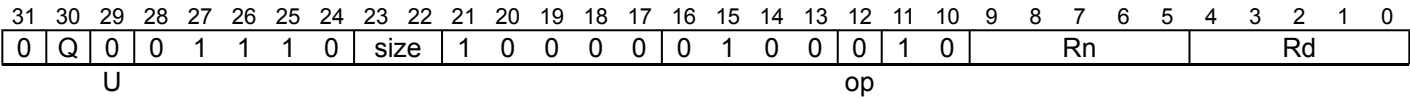
CMGT <V><d>, <V><n>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Vector



CMGT <Vd>.<T>, <Vn>.<T>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Assembler Symbols

<V> Is a width specifier, encoded in “size”:



size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

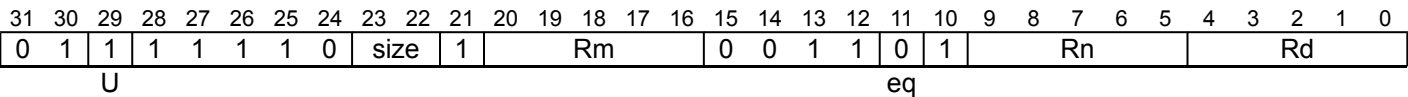
CMHI (register)

Compare unsigned Higher (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

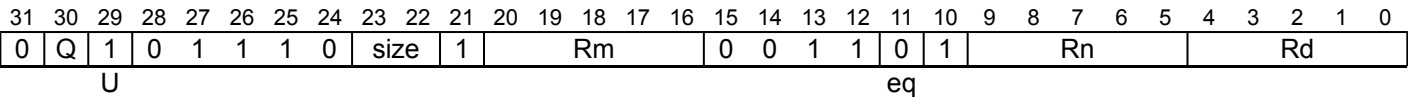
Scalar



CMHI <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Vector



CMHI <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

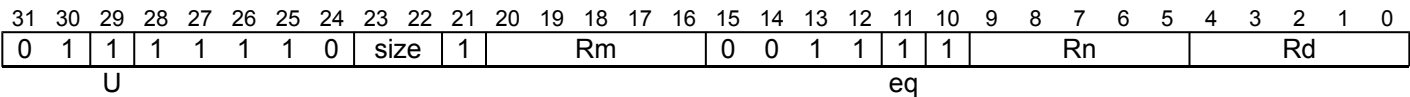
CMHS (register)

Compare unsigned Higher or Same (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than or equal to the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

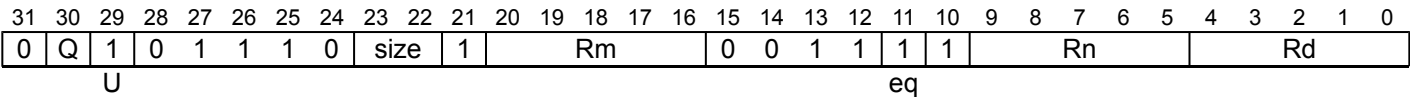
Scalar



CMHS [<V><d>](#), [<V><n>](#), [<V><m>](#)

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Vector



CMHS [<Vd>.<T>](#), [<Vn>.<T>](#), [<Vm>.<T>](#)

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Assembler Symbols

[<V>](#) Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

- [<d>](#) Is the number of the SIMD&FP destination register, in the "Rd" field.
- [<n>](#) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- [<m>](#) Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

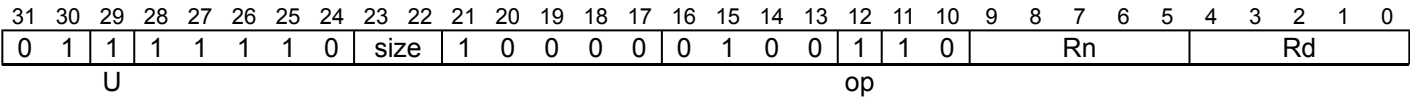
## CMLE (zero)

Compare signed Less than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar



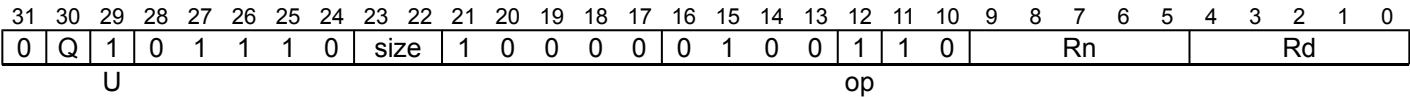
CMLE <V><d>, <V><n>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Vector



CMLE <Vd>.<T>, <Vn>.<T>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMLT (zero)

Compare signed Less than zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR EL1, CPTR EL2, and CPTR EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	1	0	0	0	0	0	0	1	0	1	0	1	0	1	0	Rn				Rd			

CMLT <V><d>, <V><n>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	0	1	0	1	0	1	0	Rn				Rd					

CMLT <Vd>.<T>, <Vn>.<T>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:



size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

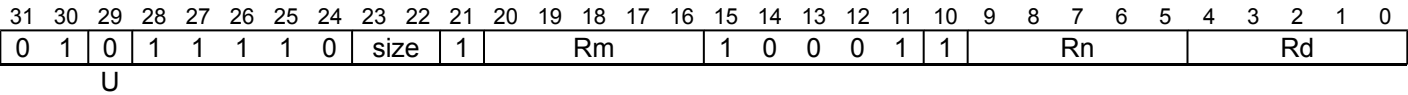
CMTST

Compare bitwise Test bits nonzero (vector). This instruction reads each vector element in the first source SIMD&FP register, performs an AND with the corresponding vector element in the second source SIMD&FP register, and if the result is not zero, sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR EL1, CPTR EL2, and CPTR EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

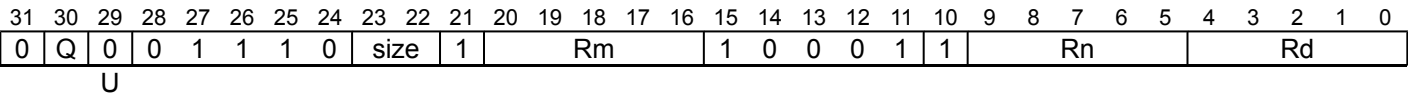
Scalar



CMTST <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');
```

Vector



CMTST <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if and_test then
        test_passed = !IsZero(element1 AND element2);
    else
        test_passed = (element1 == element2);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CNT

Population Count per byte. This instruction counts the number of bits that have a value of one in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	0	0	1	0	1	1	0	Rn				Rd					

CNT <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '00' then UNDEFINED;
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

integer count;
for e = 0 to elements-1
    count = BitCount(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<size-1:0>;
V[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

DUP (element)

Duplicate vector element to vector or scalar. This instruction duplicates the vector element at the specified element index in the source SIMD&FP register into a scalar or each element in a vector, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (scalar).

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0	imm5					0	0	0	0	0	1	Rn					Rd				

DUP <V><d>, <Vn>.<T>[<index>]

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if imm5<4> == '1' then 128 else 64;

integer esize = 8 << size;
integer datasize = esize;
integer elements = 1;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	0	imm5					0	0	0	0	0	1	Rn					Rd				

DUP <Vd>.<T>, <Vn>.<Ts>[<index>]

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if imm5<4> == '1' then 128 else 64;

if size == 3 && Q == '0' then UNDEFINED;
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<T> For the scalar variant: is the element width specifier, encoded in “imm5”:

imm5	<T>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

For the vector variant: is an arrangement specifier, encoded in "imm5:Q":

imm5	Q	<T>
x0000	x	RESERVED
xxxx1	0	8B
xxxx1	1	16B
xxx10	0	4H
xxx10	1	8H
xx100	0	2S
xx100	1	4S
x1000	0	RESERVED
x1000	1	2D

<Ts> Is an element size specifier, encoded in "imm5":

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<V> Is the destination width specifier, encoded in "imm5":

imm5	<V>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index> Is the element index encoded in "imm5":

imm5	<index>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n, idxdsize];
bits(datasize) result;
bits(esize) element;

element = Elem[operand, index, esize];
for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d, datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DUP (general)

Duplicate general-purpose register to vector. This instruction duplicates the contents of the source general-purpose register into a scalar or each element in a vector, and writes the result to the SIMD&FP destination register. Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	0	imm5					0	0	0	0	1	1	Rn					Rd				

DUP <Vd>.<T>, <R><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

// imm5<4:size+1> is IGNORED

if size == 3 && Q == '0' then UNDEFINED;
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "imm5:Q":

imm5	Q	<T>
x0000	x	RESERVED
xxxx1	0	8B
xxxx1	1	16B
xxx10	0	4H
xxx10	1	8H
xx100	0	2S
xx100	1	4S
x1000	0	RESERVED
x1000	1	2D

<R> Is the width specifier for the general-purpose source register, encoded in "imm5":

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

Unspecified bits in "imm5" are ignored but should be set to zero by an assembler.

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n, esize];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d, datasize] = result;
```



## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

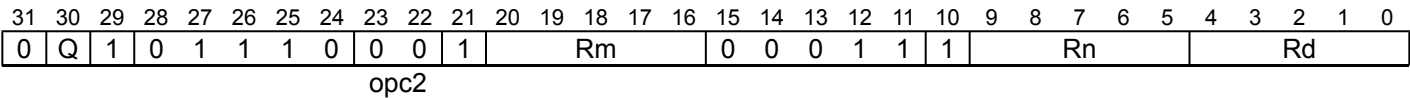
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EOR (vector)

Bitwise Exclusive-OR (vector). This instruction performs a bitwise exclusive-OR operation between the two source SIMD&FP registers, and places the result in the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



EOR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n, datasize];

operand1 = V[m, datasize];
operand2 = Zeros(datasize);
operand3 = Ones(datasize);
V[d, datasize] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```

Operational information

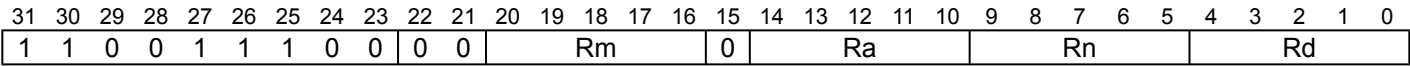
- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

EOR3

Three-way Exclusive-OR performs a three-way exclusive-OR of the values in the three source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when *FEAT\_SHA3* is implemented.

Advanced SIMD  
(FEAT\_SHA3)



EOR3 <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B

```
if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Va> Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Va = V[a, 128];
V[d, 128] = Vn EOR Vm EOR Va;
```

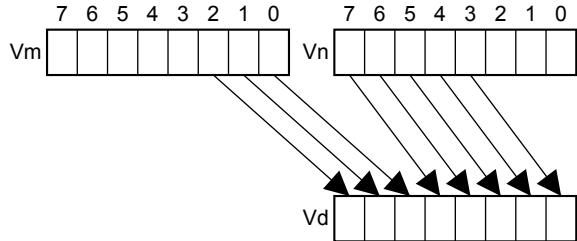
Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

EXT

Extract vector from pair of vectors. This instruction extracts the lowest vector elements from the second source SIMD&FP register and the highest vector elements from the first source SIMD&FP register, concatenates the results into a vector, and writes the vector to the destination SIMD&FP register vector. The index value specifies the lowest vector element to extract from the first source register, and consecutive elements are extracted from the first, then second, source registers until the destination vector is filled.

The following figure shows an example of the operation of EXT doubleword operation for Q = 0 and imm4<2:0> = 3.



Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	0	0	Rm				0	imm4				0	Rn				Rd						

EXT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<index>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if Q == '0' && imm4<3> == '1' then UNDEFINED;

integer datasize = if Q == '1' then 128 else 64;
integer position = UInt(imm4) << 3;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<index> Is the lowest numbered byte element to be extracted, encoded in "Q:imm4":

Q	imm4<3>	<index>
0	0	imm4<2:0>
0	1	RESERVED
1	x	imm4

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) hi = V[m, datasize];
bits(datasize) lo = V[n, datasize];
bits(datasize*2) concat = hi:lo;

V[d, datasize] = concat<(position+datasize)-1:position>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FABD

Floating-point Absolute Difference (vector). This instruction subtracts the floating-point values in the elements of the second source SIMD&FP register, from the corresponding floating-point values in the elements of the first source SIMD&FP register, places the absolute value of each result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

Scalar half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	0	Rm				0	0	0	1	0	1	Rn				Rd						

FABD <Hd> , <Hn> , <Hm>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	Rm					1	1	0	1	0	1	Rn					Rd				

FABD <V><d> , <V><n> , <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;
```

Vector half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	0	Rm				0	0	0	1	0	1	Rn				Rd						

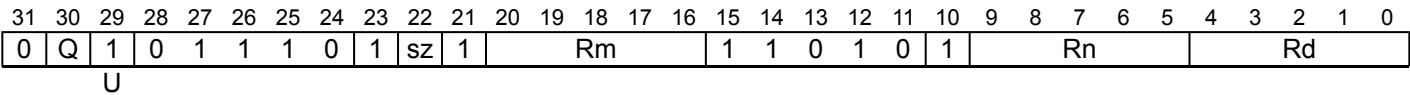
U

FABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

Vector single-precision and double-precision



FABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
bits(esize) diff;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, fpcr);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

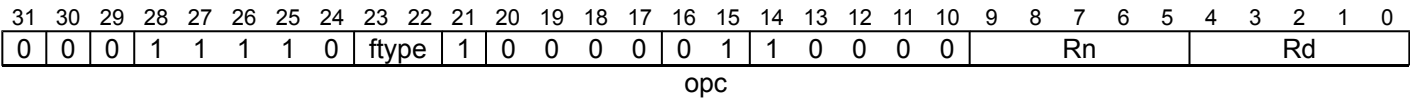
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



FABS (scalar)

Floating-point Absolute value (scalar). This instruction calculates the absolute value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

FABS <Hd>, <Hn>

Single-precision (ftype == 00)

FABS <Sd>, <Sn>

Double-precision (ftype == 01)

FABS <Dd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else 0<127:0>;

bits(esize) operand = V[n, esize];

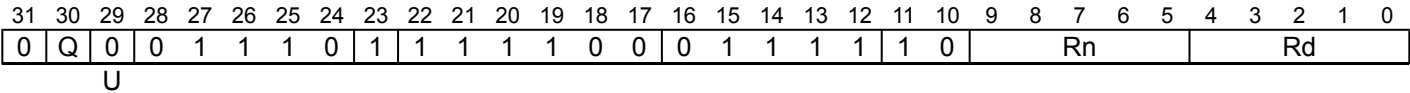
Elem[result, 0, esize] = FPAbs(operand);
V[d, 128] = result;
```



FABS (vector)

Floating-point Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(FEAT\_FP16)



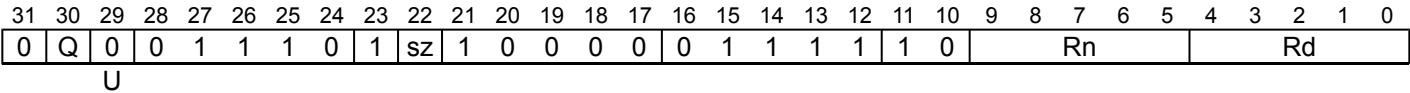
FABS <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Single-precision and double-precision



FABS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;

V[d, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FACGE

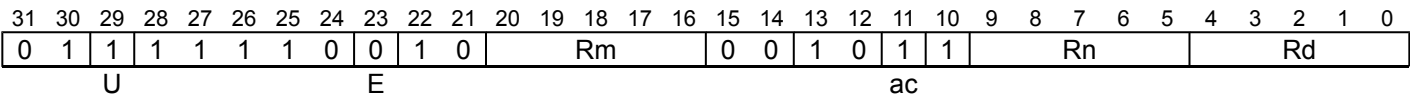
Floating-point Absolute Compare Greater than or Equal (vector). This instruction compares the absolute value of each floating-point value in the first source SIMD&FP register with the absolute value of the corresponding floating-point value in the second source SIMD&FP register and if the first value is greater than or equal to the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

Scalar half precision  
(FEAT\_FP16)



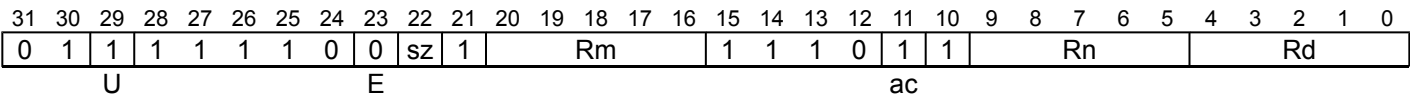
FACGE <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

Scalar single-precision and double-precision

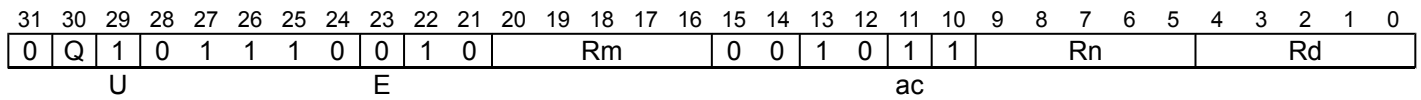


**FACGE** <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

## Vector half precision (FEAT\_FP16)



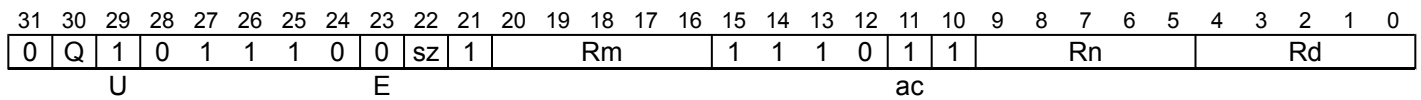
**FACGE** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

## Vector single-precision and double-precision



FACGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[m, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



FACGT

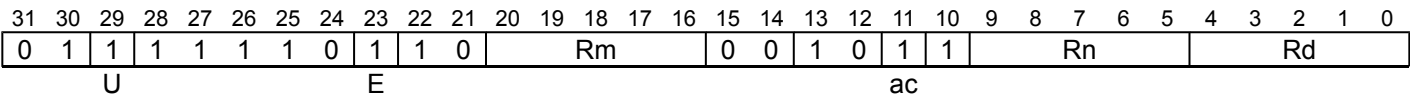
Floating-point Absolute Compare Greater than (vector). This instruction compares the absolute value of each vector element in the first source SIMD&FP register with the absolute value of the corresponding vector element in the second source SIMD&FP register and if the first value is greater than the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

Scalar half precision  
(FEAT\_FP16)



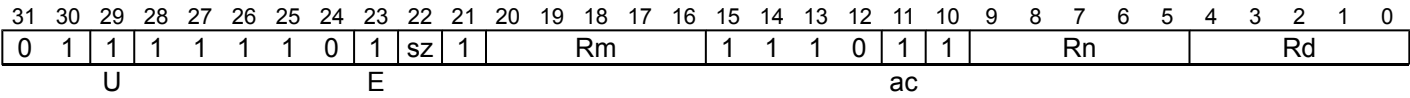
FACGT <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

Scalar single-precision and double-precision

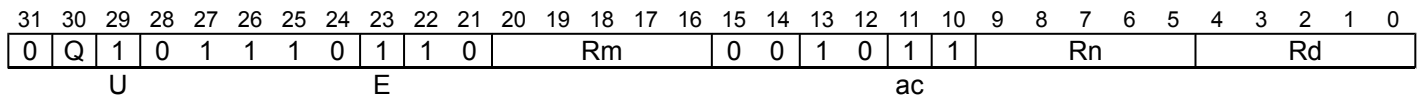


**FACGT** <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

## Vector half precision (FEAT\_FP16)



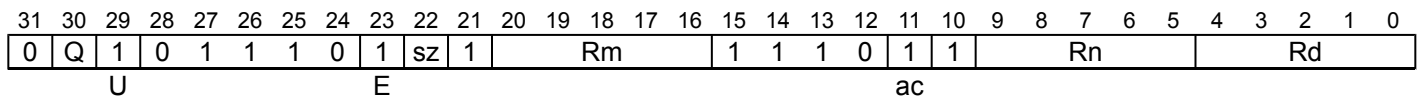
**FACGT** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

## Vector single-precision and double-precision



FACGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[m, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

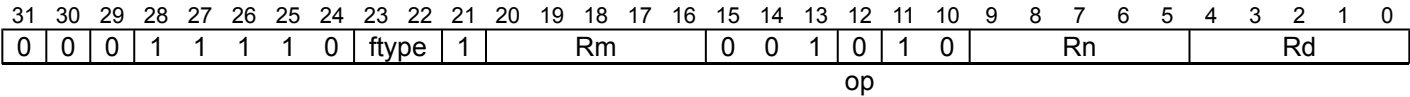
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FADD (scalar)

Floating-point Add (scalar). This instruction adds the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

```
FADD <Hd>, <Hn>, <Hm>
```

Single-precision (ftype == 00)

```
FADD <Sd>, <Sn>, <Sm>
```

Double-precision (ftype == 01)

```
FADD <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPEnabled64();  
bits(esize) operand1 = V[n, esize];  
bits(esize) operand2 = V[m, esize];  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[n, 128] else Zeros(128);  
  
Elem[result, 0, esize] = FPAdd(operand1, operand2, fpcr);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FADD (vector)

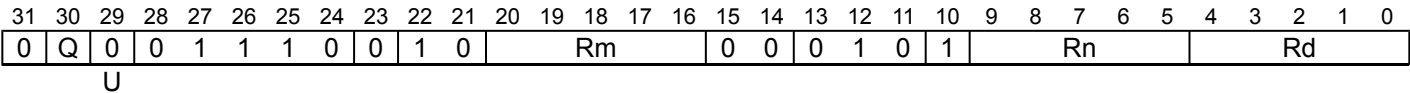
Floating-point Add (vector). This instruction adds corresponding vector elements in the two source SIMD&FP registers, writes the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(FEAT\_FP16)



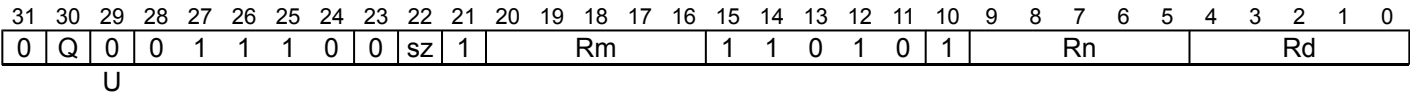
FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

Single-precision and double-precision



FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);
V[d, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



FADDP (scalar)

Floating-point Add Pair of elements (scalar). This instruction adds two floating-point vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Half-precision* and *Single-precision and double-precision*

Half-precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	1	1	0	Rn				Rd					

```
FADDP <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	1	1	0	Rn				Rd					

```
FADDP <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
```

Assembler Symbols

- <V>

For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	H
1	RESERVED
- For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D
- <d>

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <T>

For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;T&gt;</b>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “SZ”:

<b>sz</b>	<b>&lt;T&gt;</b>
0	2S
1	2D

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_FADD, operand, esize);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FADDP (vector)

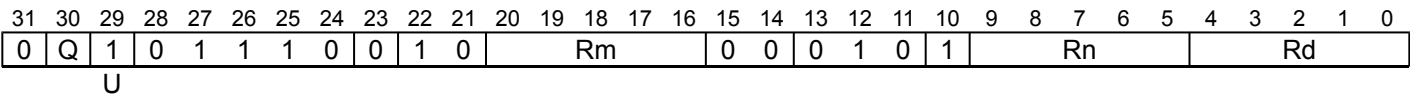
Floating-point Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(FEAT\_FP16)



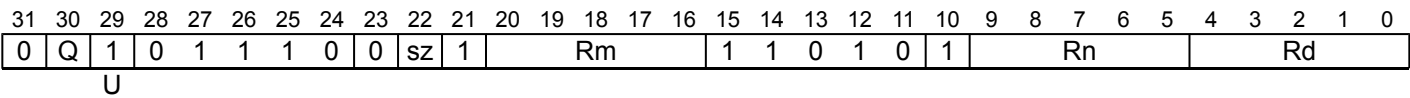
```
FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

Single-precision and double-precision



```
FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAAdd(element1, element2, FPCR[]);
V[d, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCADD

Floating-point Complex Add.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 90 or 270 degrees.
- The rotated complex number is added to the complex number from the first source register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector  
(FEAT\_FCMA)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	0	Rm			1	1	1	rot	0	1	Rn				Rd								

FCADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>

```
if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then UNDEFINED;
if Q == '0' && size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<rotate> Is the rotation, encoded in "rot":

rot	<rotate>
0	90
1	270

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element3;

for e = 0 to (elements DIV 2)-1
    case rot of
        when '0'
            element1 = FPNeg(Elem[operand2, e*2+1, esize]);
            element3 = Elem[operand2, e*2, esize];
        when '1'
            element1 = Elem[operand2, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, e*2, esize]);
    Elem[result, e*2, esize] = FPAdd(Elem[operand1, e*2, esize], element1, FPCR[]);
    Elem[result, e*2+1, esize] = FPAdd(Elem[operand1, e*2+1, esize], element3, FPCR[]);
V[d, datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# FCCMP

Floating-point Conditional quiet Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the *PSTATE*.{N, Z, C, V} flags. If the condition does not pass then the *PSTATE*.{N, Z, C, V} flags are set to the flag bit specifier.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	ftype	1	Rm				cond				0	1	Rn				0	nzcvcv						
op																															

## Half-precision (ftype == 11) (FEAT\_FP16)

FCCMP <Hn>, <Hm>, #<nzcvcv>, <cond>

## Single-precision (ftype == 00)

FCCMP <Sn>, <Sm>, #<nzcvcv>, <cond>

## Double-precision (ftype == 01)

FCCMP <Dn>, <Dm>, #<nzcvcv>, <cond>

```
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UNDEFINED;

bits(4) flags = nzcvcv;
```

## Assembler Symbols

- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <nzcvcv> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcvcv" field.
- <cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

## Operation

```
CheckFPEnabled64();  
  
bits(datasize) operand1 = V[n, datasize];  
bits(datasize) operand2;  
  
operand2 = V[m, datasize];  
  
if ConditionHolds(cond) then  
    flags = FPCompare(operand1, operand2, FALSE, FPCR[]);  
PSTATE.<N,Z,C,V> = flags;
```

## Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the [PSTATE](#) condition flags to N=0, Z=0, C=1, and V=1.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# FCCMPE

Floating-point Conditional signaling Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the *PSTATE*.{N, Z, C, V} flags. If the condition does not pass then the *PSTATE*.{N, Z, C, V} flags are set to the flag bit specifier.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	f	t	y	p	e	1	Rm			cond			0	1	Rn			1	nzc				v	
																														op	

## Half-precision (ftype == 11) (FEAT\_FP16)

FCCMPE <Hn>, <Hm>, #<nzc>, <cond>

## Single-precision (ftype == 00)

FCCMPE <Sn>, <Sm>, #<nzc>, <cond>

## Double-precision (ftype == 01)

FCCMPE <Dn>, <Dm>, #<nzc>, <cond>

```
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UNDEFINED;

bits(4) flags = nzc;
```

## Assembler Symbols

- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <nzc> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzc" field.
- <cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

## Operation

```
CheckFPEnabled64();  
  
bits(datasize) operand1 = V[n, datasize];  
bits(datasize) operand2;  
  
operand2 = V[m, datasize];  
  
if ConditionHolds(cond) then  
    flags = FPCompare(operand1, operand2, TRUE, FPCR[]);  
PSTATE.<N,Z,C,V> = flags;
```

## Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the [PSTATE](#) condition flags to N=0, Z=0, C=1, and V=1.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCMEQ (register)

Floating-point Compare Equal (vector). This instruction compares each floating-point value from the first source SIMD&FP register, with the corresponding floating-point value from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

Scalar half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	1	0	Rm				0	0	1	0	0	1	Rn				Rd						
U								E				ac																			

FCMEQ <Hd> , <Hn> , <Hm>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	Rm				1	1	1	0	0	1	Rn				Rd						
U								E				ac																			

FCMEQ <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

## Vector half precision (FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	0	Rm				0	0	1	0	0	1	Rn				Rd						
U								E								ac															

FCMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

## Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	Rm				1	1	1	0	0	1	Rn				Rd						
U								E								ac															

FCMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

### Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[m, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCMEQ (zero)

Floating-point Compare Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	1	1	0	Rn				Rd					
U																op															

**FCMEQ** <Hd>, <Hn>, #0.0

```
if !HaveFP16Ext() then UNDEFINED;
```

```
integer d = UInt(Rd);
```

```
integer n = UInt(Rn);
```

```
integer esize = 16;
```

```
integer datasize = esize;
```

```
integer elements = 1;
```

```
CompareOp comparison;
```

```
case op:U of
```

```
  when '00' comparison = CompareOp\_GT;
```

```
  when '01' comparison = CompareOp\_GE;
```

```
  when '10' comparison = CompareOp\_EQ;
```

```
  when '11' comparison = CompareOp\_LE;
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	1	1	0	Rn				Rd					
U																op															

**FCMEQ** <V><d>, <V><n>, #0.0

```
integer d = UInt(Rd);
```

```
integer n = UInt(Rn);
```

```
integer esize = 32 << UInt(sz);
```

```
integer datasize = esize;
```

```
integer elements = 1;
```

```
CompareOp comparison;
```

```
case op:U of
```

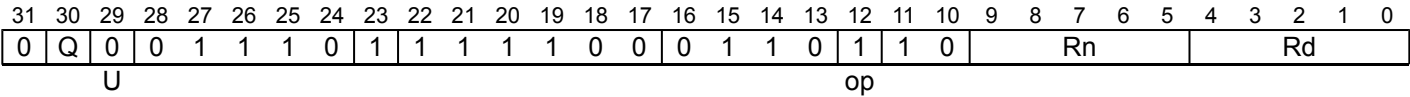
```
  when '00' comparison = CompareOp\_GT;
```

```
  when '01' comparison = CompareOp\_GE;
```

```
  when '10' comparison = CompareOp\_EQ;
```

```
  when '11' comparison = CompareOp\_LE;
```

Vector half precision  
(FEAT\_FP16)



FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0

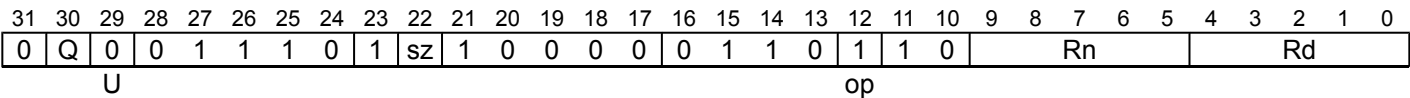
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision



FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D
- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.



<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) zero = FPZero('0', esize);
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCMGE (register)

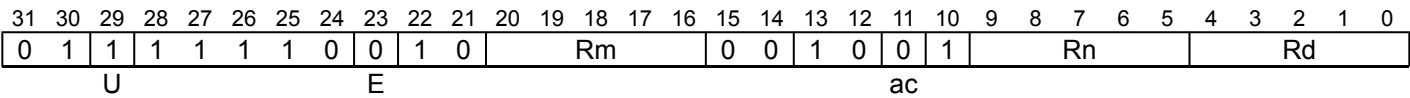
Floating-point Compare Greater than or Equal (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than or equal to the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

Scalar half precision  
(FEAT\_FP16)



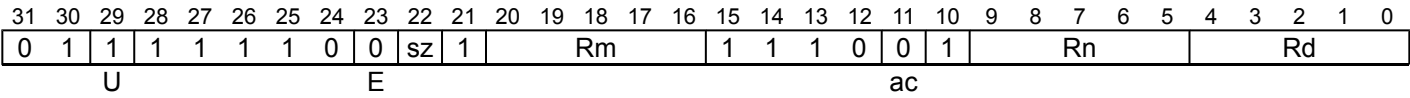
FCMGE <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

Scalar single-precision and double-precision

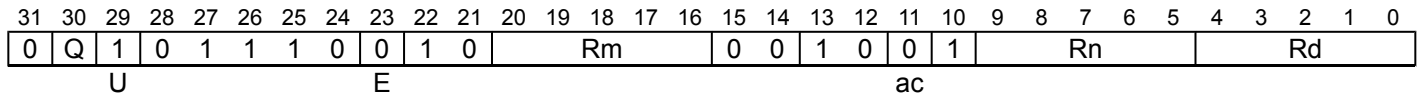


FCMGE <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

## Vector half precision (FEAT\_FP16)



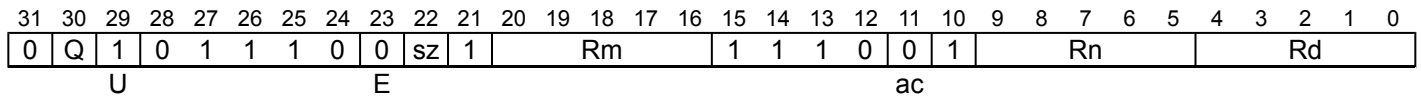
FCMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

## Vector single-precision and double-precision



FCMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[m, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCMGE (zero)

Floating-point Compare Greater than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#), [Scalar single-precision and double-precision](#), [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	0	1	0	Rn				Rd					
U																op															

**FCMGE** <Hd>, <Hn>, #0.0

```
if !HaveFP16Ext() then UNDEFINED;
```

```
integer d = UInt(Rd);
```

```
integer n = UInt(Rn);
```

```
integer esize = 16;
```

```
integer datasize = esize;
```

```
integer elements = 1;
```

```
CompareOp comparison;
```

```
case op:U of
```

```
    when '00' comparison = CompareOp_GT;
```

```
    when '01' comparison = CompareOp_GE;
```

```
    when '10' comparison = CompareOp_EQ;
```

```
    when '11' comparison = CompareOp_LE;
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
U																op															

**FCMGE** <V><d>, <V><n>, #0.0

```
integer d = UInt(Rd);
```

```
integer n = UInt(Rn);
```

```
integer esize = 32 << UInt(sz);
```

```
integer datasize = esize;
```

```
integer elements = 1;
```

```
CompareOp comparison;
```

```
case op:U of
```

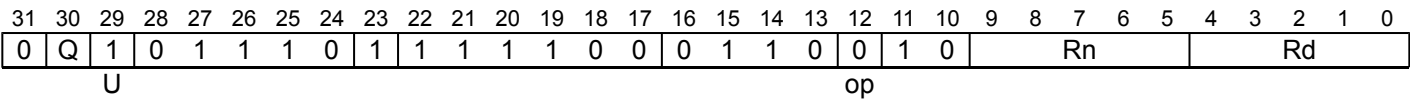
```
    when '00' comparison = CompareOp_GT;
```

```
    when '01' comparison = CompareOp_GE;
```

```
    when '10' comparison = CompareOp_EQ;
```

```
    when '11' comparison = CompareOp_LE;
```

Vector half precision  
(FEAT\_FP16)



FCMGE <Vd>.<T>, <Vn>.<T>, #0.0

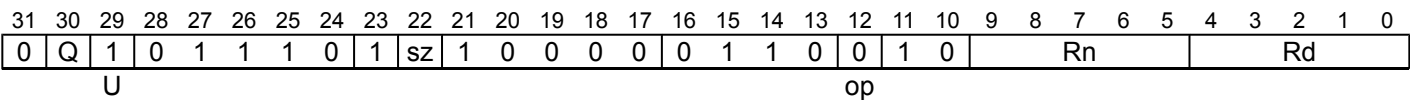
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision



FCMGE <Vd>.<T>, <Vn>.<T>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D
- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) zero = FPZero('0', esize);
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



FCMGT (register)

Floating-point Compare Greater than (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

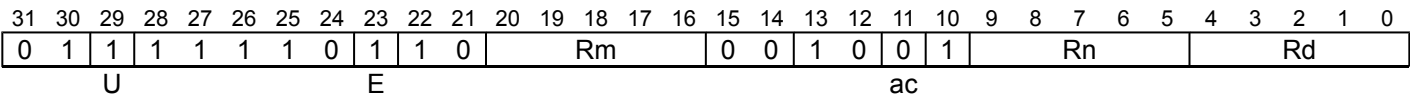
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

Scalar half precision

(FEAT\_FP16)



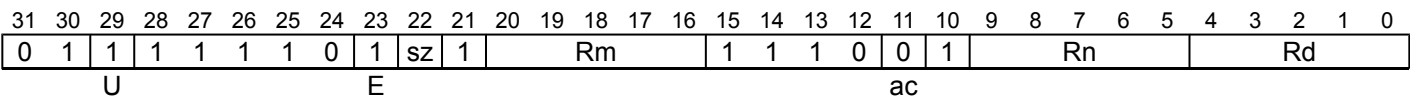
FCMGT <Hd> , <Hn> , <Hm>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

Scalar single-precision and double-precision



FCMGT <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

## Vector half precision (FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	0	Rm				0	0	1	0	0	1	Rn				Rd						
U								E								ac															

FCMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

## Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	Rm					1	1	1	0	0	1	Rn					Rd				
U								E								ac															

FCMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

## Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[m, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCMGT (zero)

Floating-point Compare Greater than zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	0	1	0	Rn				Rd					
U																op															

**FCMGT** <Hd>, <Hn>, #0.0

```
if !HaveFP16Ext() then UNDEFINED;
```

```
integer d = UInt(Rd);
```

```
integer n = UInt(Rn);
```

```
integer esize = 16;
```

```
integer datasize = esize;
```

```
integer elements = 1;
```

```
CompareOp comparison;
```

```
case op:U of
```

```
  when '00' comparison = CompareOp\_GT;
```

```
  when '01' comparison = CompareOp\_GE;
```

```
  when '10' comparison = CompareOp\_EQ;
```

```
  when '11' comparison = CompareOp\_LE;
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
U																op															

**FCMGT** <V><d>, <V><n>, #0.0

```
integer d = UInt(Rd);
```

```
integer n = UInt(Rn);
```

```
integer esize = 32 << UInt(sz);
```

```
integer datasize = esize;
```

```
integer elements = 1;
```

```
CompareOp comparison;
```

```
case op:U of
```

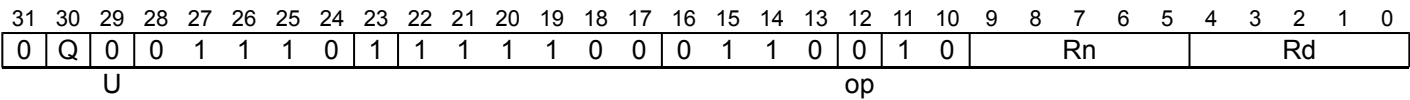
```
  when '00' comparison = CompareOp\_GT;
```

```
  when '01' comparison = CompareOp\_GE;
```

```
  when '10' comparison = CompareOp\_EQ;
```

```
  when '11' comparison = CompareOp\_LE;
```

Vector half precision  
(FEAT\_FP16)



FCMGT <Vd>.<T>, <Vn>.<T>, #0.0

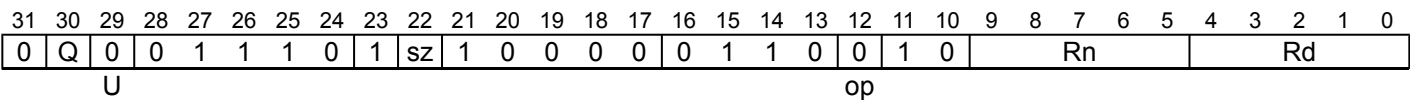
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision



FCMGT <Vd>.<T>, <Vn>.<T>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D
- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) zero = FPZero('0', esize);
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCMLA

Floating-point Complex Multiply Accumulate.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers and the destination register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding. This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector  
(FEAT\_FCMA)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	Q	1	0	1	1	1	0	size	0	Rm						1	1	0	rot	1	Rn						Rd					

FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>

```
if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then UNDEFINED;
if Q == '0' && size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<rotate> Is the rotation, encoded in "rot":



rot	<rotate>
00	0
01	90
10	180
11	270

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) element3;
bits(esize) element4;
FPCRType fpcr = FPCR[];

for e = 0 to (elements DIV 2)-1
    case rot of
        when '00'
            element1 = Elem[operand2, e*2, esize];
            element2 = Elem[operand1, e*2, esize];
            element3 = Elem[operand2, e*2+1, esize];
            element4 = Elem[operand1, e*2, esize];
        when '01'
            element1 = FPNeg(Elem[operand2, e*2+1, esize]);
            element2 = Elem[operand1, e*2+1, esize];
            element3 = Elem[operand2, e*2, esize];
            element4 = Elem[operand1, e*2+1, esize];
        when '10'
            element1 = FPNeg(Elem[operand2, e*2, esize]);
            element2 = Elem[operand1, e*2, esize];
            element3 = FPNeg(Elem[operand2, e*2+1, esize]);
            element4 = Elem[operand1, e*2, esize];
        when '11'
            element1 = Elem[operand2, e*2+1, esize];
            element2 = Elem[operand1, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, e*2, esize]);
            element4 = Elem[operand1, e*2+1, esize];

    Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, fpcr);
    Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, fpcr);

V[d, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCMLA (by element)

Floating-point Complex Multiply Accumulate (by element).

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on complex numbers from the first source register and the destination register with the specified complex number from the second source register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding. This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Vector (FEAT\_FCMA)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size	L	M			Rm		0	rot	1	H	0												Rd

(size == 01)

```
FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>
```

(size == 10)

```
FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>
```

```
if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index;
if size == '00' || size == '11' then UNDEFINED;
if size == '01' then index = UInt(H:L);
if size == '10' then index = UInt(H);
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
if size == '10' && (L == '1' || Q == '0') then UNDEFINED;
if size == '01' && H == '1' && Q == '0' then UNDEFINED;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in "size:H:L":

size	<index>
00	RESERVED
01	H:L
10	H
11	RESERVED

<rotate> Is the rotation, encoded in "rot":

rot	<rotate>
00	0
01	90
10	180
11	270

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
FPCRTYPE fpcr = FPCR[];

for e = 0 to (elements DIV 2)-1
    bits(esize) element1;
    bits(esize) element2;
    bits(esize) element3;
    bits(esize) element4;
    case rot of
        when '00'
            element1 = Elem[operand2, index*2, esize];
            element2 = Elem[operand1, e*2, esize];
            element3 = Elem[operand2, index*2+1, esize];
            element4 = Elem[operand1, e*2, esize];
        when '01'
            element1 = FPNeg(Elem[operand2, index*2+1, esize]);
            element2 = Elem[operand1, e*2+1, esize];
            element3 = Elem[operand2, index*2, esize];
            element4 = Elem[operand1, e*2+1, esize];
        when '10'
            element1 = FPNeg(Elem[operand2, index*2, esize]);
            element2 = Elem[operand1, e*2, esize];
            element3 = FPNeg(Elem[operand2, index*2+1, esize]);
            element4 = Elem[operand1, e*2, esize];
        when '11'
            element1 = Elem[operand2, index*2+1, esize];
            element2 = Elem[operand1, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, index*2, esize]);
            element4 = Elem[operand1, e*2+1, esize];

    Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, fpcr);
    Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, fpcr);

V[d, datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCMLE (zero)

Floating-point Compare Less than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#), [Scalar single-precision and double-precision](#), [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	0	1	1	0	Rn				Rd					
U										op																					

**FCMLE** <Hd>, <Hn>, #0.0

```
if !HaveFP16Ext() then UNDEFINED;
```

```
integer d = UInt(Rd);
```

```
integer n = UInt(Rn);
```

```
integer esize = 16;
```

```
integer datasize = esize;
```

```
integer elements = 1;
```

```
CompareOp comparison;
```

```
case op:U of
```

```
    when '00' comparison = CompareOp\_GT;
```

```
    when '01' comparison = CompareOp\_GE;
```

```
    when '10' comparison = CompareOp\_EQ;
```

```
    when '11' comparison = CompareOp\_LE;
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	0	1	1	0	Rn				Rd					
U										op																					

**FCMLE** <V><d>, <V><n>, #0.0

```
integer d = UInt(Rd);
```

```
integer n = UInt(Rn);
```

```
integer esize = 32 << UInt(sz);
```

```
integer datasize = esize;
```

```
integer elements = 1;
```

```
CompareOp comparison;
```

```
case op:U of
```

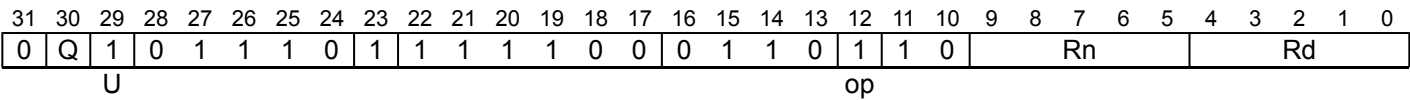
```
    when '00' comparison = CompareOp\_GT;
```

```
    when '01' comparison = CompareOp\_GE;
```

```
    when '10' comparison = CompareOp\_EQ;
```

```
    when '11' comparison = CompareOp\_LE;
```

Vector half precision  
(FEAT\_FP16)



FCMLE <Vd>.<T>, <Vn>.<T>, #0.0

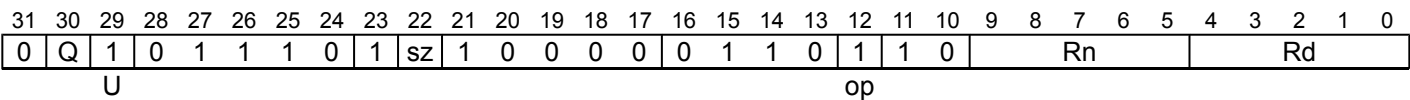
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision



FCMLE <Vd>.<T>, <Vn>.<T>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D
- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) zero = FPZero('0', esize);
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCMLT (zero)

Floating-point Compare Less than zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	0	1	0	Rn				Rd					

**FCMLT** <Hd>, <Hn>, #0.0

```
if !HaveFP16Ext() then UNDEFINED;
```

```
integer d = UInt(Rd);
```

```
integer n = UInt(Rn);
```

```
integer esize = 16;
```

```
integer datasize = esize;
```

```
integer elements = 1;
```

```
CompareOp comparison = CompareOp\_LT;
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	1	0	1	0	Rn				Rd					

**FCMLT** <V><d>, <V><n>, #0.0

```
integer d = UInt(Rd);
```

```
integer n = UInt(Rn);
```

```
integer esize = 32 << UInt(sz);
```

```
integer datasize = esize;
```

```
integer elements = 1;
```

```
CompareOp comparison = CompareOp\_LT;
```

### Vector half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	0	1	0	Rn				Rd					



FCMLT <Vd>.<T>, <Vn>.<T>, #0.0

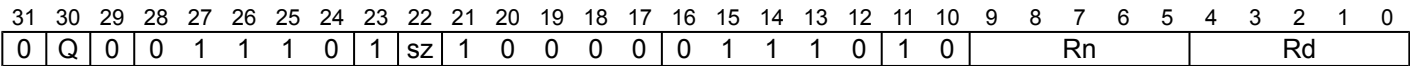
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
```

Vector single-precision and double-precision



FCMLT <Vd>.<T>, <Vn>.<T>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) zero = FPZero('0', esize);
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCMP

Floating-point quiet Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the *PSTATE*.{N, Z, C, V} flags.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	ftype	1	Rm				0	0	1	0	0	0	Rn				0	x	0	0	0			
opc																															

Half-precision (ftype == 11 && opc == 00)  
(FEAT\_FP16)

```
FCMP <Hn>, <Hm>
```

Half-precision, zero (ftype == 11 && Rm == (00000) && opc == 01)  
(FEAT\_FP16)

```
FCMP <Hn>, #0.0
```

Single-precision (ftype == 00 && opc == 00)

```
FCMP <Sn>, <Sm>
```

Single-precision, zero (ftype == 00 && Rm == (00000) && opc == 01)

```
FCMP <Sn>, #0.0
```

Double-precision (ftype == 01 && opc == 00)

```
FCMP <Dn>, <Dm>
```

Double-precision, zero (ftype == 01 && Rm == (00000) && opc == 01)

```
FCMP <Dn>, #0.0
```

```
integer n = UInt(Rn);
integer m = UInt(Rm);    // ignored when opc<0> == '1'

integer datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UNDEFINED;

boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');
```

## Assembler Symbols

<Dn>	For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.  For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn>	For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.  For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn>	For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.  For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPEnabled64();  
  
bits(datasize) operand1 = V[n, datasize];  
bits(datasize) operand2;  
  
operand2 = if cmp_with_zero then FPZero('0', datasize) else V[m, datasize];  
  
PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR[]);
```

## Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the *PSTATE* condition flags to N=0, Z=0, C=1, and V=1.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCMPE

Floating-point signaling Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the *PSTATE*.{N, Z, C, V} flags. This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN. A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*. Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	1	1	1	1	0	ftype	1	Rm						0	0	1	0	0	0	Rn						1	x	0	0	0
																											opc					

Half-precision (ftype == 11 && opc == 10)  
(FEAT\_FP16)

```
FCMPE <Hn>, <Hm>
```

Half-precision, zero (ftype == 11 && Rm == (00000) && opc == 11)  
(FEAT\_FP16)

```
FCMPE <Hn>, #0.0
```

Single-precision (ftype == 00 && opc == 10)

```
FCMPE <Sn>, <Sm>
```

Single-precision, zero (ftype == 00 && Rm == (00000) && opc == 11)

```
FCMPE <Sn>, #0.0
```

Double-precision (ftype == 01 && opc == 10)

```
FCMPE <Dn>, <Dm>
```

Double-precision, zero (ftype == 01 && Rm == (00000) && opc == 11)

```
FCMPE <Dn>, #0.0
```

```
integer n = UInt(Rn);
integer m = UInt(Rm);    // ignored when opc<0> == '1'

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');
```

## Assembler Symbols

<Dn>	For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.  For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn>	For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.  For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn>	For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.  For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPEnabled64();  
  
bits(datasize) operand1 = V[n, datasize];  
bits(datasize) operand2;  
  
operand2 = if cmp_with_zero then FPZero('0', datasize) else V[m, datasize];  
  
PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR[]);
```

## Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the *PSTATE* condition flags to N=0, Z=0, C=1, and V=1.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# FCSEL

Floating-point Conditional Select (scalar). This instruction allows the SIMD&FP destination register to take the value from either one or the other of two SIMD&FP source registers. If the condition passes, the first SIMD&FP source register value is taken, otherwise the second SIMD&FP source register value is taken.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	1	1	1	1	0	ftype	1	Rm						cond					1	1	Rn						Rd			

Half-precision (ftype == 11)  
(FEAT\_FP16)

FCSEL <Hd>, <Hn>, <Hm>, <cond>

Single-precision (ftype == 00)

FCSEL <Sd>, <Sn>, <Sm>, <cond>

Double-precision (ftype == 01)

FCSEL <Dd>, <Dn>, <Dm>, <cond>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UNDEFINED;
```

## Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

## Operation

```
CheckFPEnabled64();  
bits(datasize) result;  
  
result = if ConditionHolds(cond) then V[n, datasize] else V[m, datasize];  
  
V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# FCVT

Floating-point Convert precision (scalar). This instruction converts the floating-point value in the SIMD&FP source register to the precision for the destination register data type using the rounding mode that is determined by the [FPCR](#) and writes the result to the SIMD&FP destination register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	ftype	1	0	0	0	1	opc	1	0	0	0	0	Rn						Rd					

Half-precision to single-precision (ftype == 11 && opc == 00)

FCVT <Sd>, <Hn>

Half-precision to double-precision (ftype == 11 && opc == 01)

FCVT <Dd>, <Hn>

Single-precision to half-precision (ftype == 00 && opc == 11)

FCVT <Hd>, <Sn>

Single-precision to double-precision (ftype == 00 && opc == 01)

FCVT <Dd>, <Sn>

Double-precision to half-precision (ftype == 01 && opc == 11)

FCVT <Hd>, <Dn>

Double-precision to single-precision (ftype == 01 && opc == 00)

FCVT <Sd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer srctype;
integer dstsize;

if ftype == opc then UNDEFINED;

case ftype of
  when '00' srctype = 32;
  when '01' srctype = 64;
  when '10' UNDEFINED;
  when '11' srctype = 16;
case opc of
  when '00' dstsize = 32;
  when '01' dstsize = 64;
  when '10' UNDEFINED;
  when '11' dstsize = 16;
```

## Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.  
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
bits(srcsize) operand = V[n, srcsize];  
FPCRTType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
  
Elem[result, 0, dstsize] = FPConvert(operand, fpcr, dstsize);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

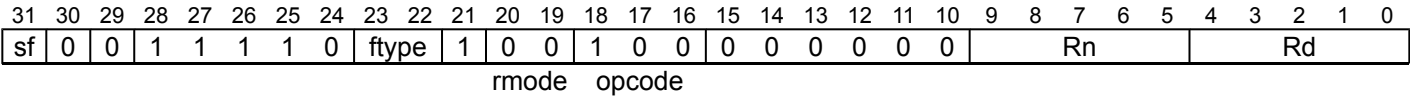
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTAS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision to 32-bit (sf == 0 && ftype == 11)  
(FEAT\_FP16)

```
FCVTAS <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)  
(FEAT\_FP16)

```
FCVTAS <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTAS <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTAS <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTAS <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTAS <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer decode_fltsize;

case ftype of
  when '00'
    decode_fltsize = 32;
  when '01'
    decode_fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      decode_fltsize = 16;
    else
      UNDEFINED;
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRTYPE fpcr = FPCR[];  
bits(decode_fltsize) fltval;  
bits(intsize) intval;  
  
fltval = V[n, decode_fltsize];  
intval = FPToFixed(fltval, 0, FALSE, fpcr, FPRounding_TIEAWAY, intsize);  
X[d, intsize] = intval;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCVTAS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to a signed integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#), [Scalar single-precision and double-precision](#), [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0	1	0	Rn				Rd					
U																															

**FCVTAS** <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn				Rd					
U																															

**FCVTAS** <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

### Vector half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0	1	0	Rn				Rd					
U																															

FCVTAS <Vd>.<T>, <Vn>.<T>

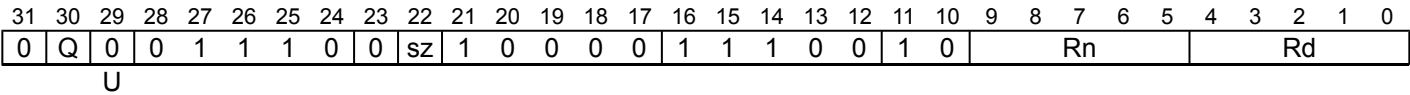
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTAS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

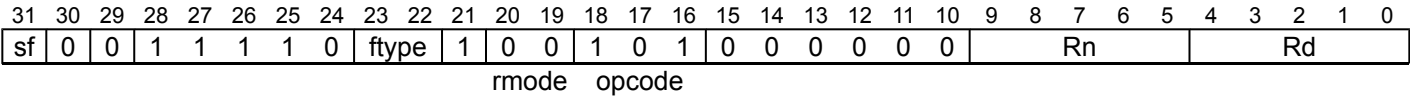
```
CheckFPAdvSIMDEnabled64();  
bits(datasize) operand = V[n, datasize];  
  
bits(esize) element;  
FPCRType fpcr = FPCR[];  
boolean merge = elements == 1 && IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
  
for e = 0 to elements-1  
    element = Elem[operand, e, esize];  
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTAU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register. A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*. Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision to 32-bit (sf == 0 && ftype == 11)  
(FEAT\_FP16)

```
FCVTAU <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)  
(FEAT\_FP16)

```
FCVTAU <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTAU <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTAU <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTAU <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTAU <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer decode_fltsize;

case ftype of
  when '00'
    decode_fltsize = 32;
  when '01'
    decode_fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      decode_fltsize = 16;
    else
      UNDEFINED;
```



## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRTYPE fpcr = FPCR[];  
bits(decode_fltsize) fltval;  
bits(intsize) intval;  
  
fltval = V[n, decode_fltsize];  
intval = FPToFixed(fltval, 0, TRUE, fpcr, FPRounding_TIEAWAY, intsize);  
X[d, intsize] = intval;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCVTAU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0	1	0	Rn				Rd					
U																															

**FCVTAU** <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn				Rd					
U																															

**FCVTAU** <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

### Vector half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0	1	0	Rn				Rd					
U																															

FCVTAU <Vd>.<T>, <Vn>.<T>

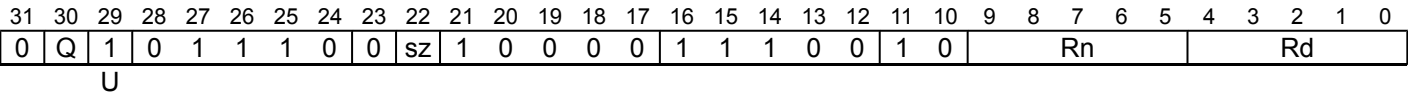
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTAU <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();  
bits(datasize) operand = V[n, datasize];  
  
bits(esize) element;  
FPCRType fpcr = FPCR[];  
boolean merge = elements == 1 && IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
  
for e = 0 to elements-1  
    element = Elem[operand, e, esize];  
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTL, FCVTL2

Floating-point Convert to higher precision Long (vector). This instruction reads each element in a vector in the SIMD&FP source register, converts each value to double the precision of the source element using the rounding mode that is determined by the *FPCR*, and writes each result to the equivalent element of the vector in the SIMD&FP destination register.

Where the operation lengthens a 64-bit vector to a 128-bit vector, the FCVTL2 variant operates on the elements in the top 64 bits of the source register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	0	1	1	1	1	0	Rn					Rd				

FCVTL{2} <Vd>.<Ta>, <Vn>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16 << UInt(sz);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “sz”:

sz	<Ta>
0	4S
1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<Tb>
0	0	4H
0	1	8H
1	0	2S
1	1	4S

## Operation

```
CheckFPAdvSIMDEnabled64();  
bits(datasize) operand = Vpart[n, part, datasize];  
bits(2*datasize) result;  
  
for e = 0 to elements-1  
    Elem[result, e, 2*esize] = FPConvert(Elem[operand, e, esize], FPCR[], 2 * esize);  
  
V[d, 2*datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCVTMS (scalar)

Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	ftype	1	1	0	0	0	0	0	0	0	0	0	0	0	Rn				Rd					
rmode												opcode																			

### Half-precision to 32-bit (sf == 0 && ftype == 11) (FEAT\_FP16)

FCVTMS <Wd>, <Hn>

### Half-precision to 64-bit (sf == 1 && ftype == 11) (FEAT\_FP16)

FCVTMS <Xd>, <Hn>

### Single-precision to 32-bit (sf == 0 && ftype == 00)

FCVTMS <Wd>, <Sn>

### Single-precision to 64-bit (sf == 1 && ftype == 00)

FCVTMS <Xd>, <Sn>

### Double-precision to 32-bit (sf == 0 && ftype == 01)

FCVTMS <Wd>, <Dn>

### Double-precision to 64-bit (sf == 1 && ftype == 01)

FCVTMS <Xd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer decode_fltsize;
FPRounding rounding;

case ftype of
  when '00'
    decode_fltsize = 32;
  when '01'
    decode_fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      decode_fltsize = 16;
    else
      UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRTYPE fpcr = FPCR[];  
bits(decode_fltsize) fltval;  
bits(intsize) intval;  
  
fltval = V[n, decode_fltsize];  
intval = FPToFixed(fltval, 0, FALSE, fpcr, rounding, intsize);  
X[d, intsize] = intval;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## FCVTMS (vector)

Floating-point Convert to Signed integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

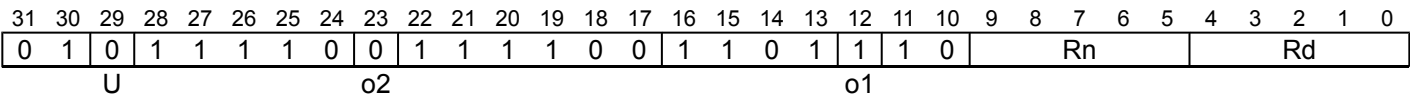
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: *Scalar half precision*, *Scalar single-precision and double-precision*, *Vector half precision* and *Vector single-precision and double-precision*

### Scalar half precision

(FEAT\_FP16)



FCVTMS <Hd>, <Hn>

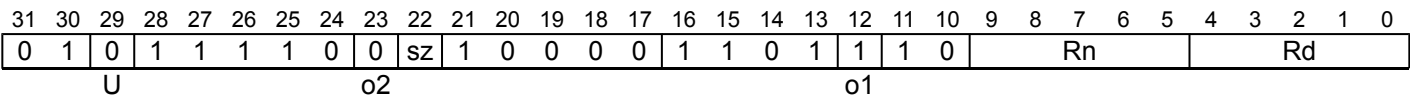
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision



FCVTMS <V><d>, <V><n>

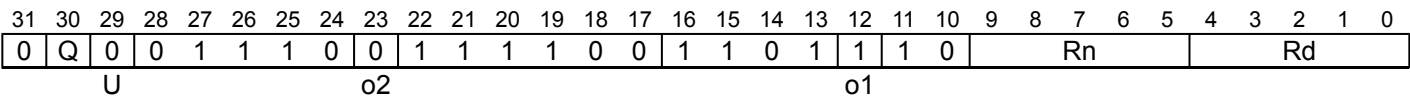
```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Vector half precision

(FEAT\_FP16)



FCVTMS <Vd>.<T>, <Vn>.<T>

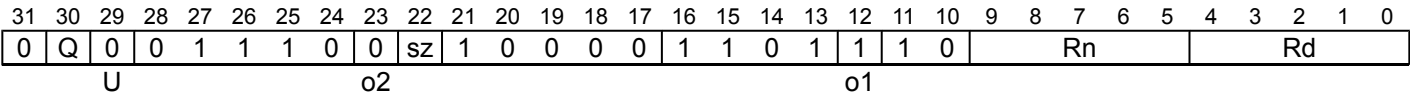
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTMS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();  
bits(datasize) operand = V[n, datasize];  
  
bits(esize) element;  
FPCRType fpcr = FPCR[];  
boolean merge = elements == 1 && IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
  
for e = 0 to elements-1  
    element = Elem[operand, e, esize];  
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

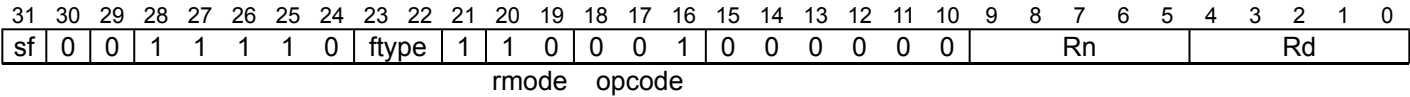
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTMU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision to 32-bit (sf == 0 && ftype == 11)  
(FEAT\_FP16)

```
FCVTMU <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)  
(FEAT\_FP16)

```
FCVTMU <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTMU <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTMU <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTMU <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTMU <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer decode_ftsize;
FPRounding rounding;

case ftype of
  when '00'
    decode_ftsize = 32;
  when '01'
    decode_ftsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      decode_ftsize = 16;
    else
      UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRTYPE fpcr = FPCR[];  
bits(decode_fltsize) fltval;  
bits(intsize) intval;  
  
fltval = V[n, decode_fltsize];  
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding, intsize);  
X[d, intsize] = intval;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCVTMU (vector)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#), [Scalar single-precision and double-precision](#), [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn				Rd					
U								o2				o1																			

**FCVTMU** <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn				Rd					
U								o2				o1																			

**FCVTMU** <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Vector half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn				Rd					
U								o2				o1																			

FCVTMU <Vd>.<T>, <Vn>.<T>

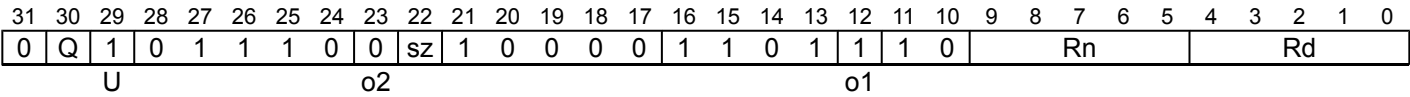
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTMU <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();  
bits(datasize) operand = V[n, datasize];  
  
bits(esize) element;  
FPCRType fpcr = FPCR[];  
boolean merge = elements == 1 && IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
  
for e = 0 to elements-1  
    element = Elem[operand, e, esize];  
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



FCVTN, FCVTN2

Floating-point Convert to lower precision Narrow (vector). This instruction reads each vector element in the SIMD&FP source register, converts each result to half the precision of the source element, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The rounding mode is determined by the *FPCR*.  
The FCVTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the FCVTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	0	0	0	0	1	0	1	1	0	1	0	Rn				Rd					

FCVTN{2} <Vd>.<Tb>, <Vn>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16 << UInt(sz);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<Tb>
0	0	4H
0	1	8H
1	0	2S
1	1	4S

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “sz”:

sz	<Ta>
0	4S
1	2D

## Operation

```
CheckFPAdvSIMDEnabled64();  
bits(2*datasize) operand = V[n, 2*datasize];  
bits(datasize) result;  
  
for e = 0 to elements-1  
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], FPCR[], esize);  
  
Vpart[d, part, datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCVTNS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	ftype	1	0	0	0	0	0	0	0	0	0	0	0	0	Rn				Rd					
rmode										opcode																					

Half-precision to 32-bit (sf == 0 && ftype == 11)  
(FEAT\_FP16)

FCVTNS <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && ftype == 11)  
(FEAT\_FP16)

FCVTNS <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 && ftype == 00)

FCVTNS <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && ftype == 00)

FCVTNS <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && ftype == 01)

FCVTNS <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && ftype == 01)

FCVTNS <Xd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer decode_ftsize;
FPRounding rounding;

case ftype of
  when '00'
    decode_ftsize = 32;
  when '01'
    decode_ftsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      decode_ftsize = 16;
    else
      UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRTyp fpcr = FPCR[];  
bits(decode_fltsize) fltval;  
bits(intsize) intval;  
  
fltval = V[n, decode_fltsize];  
intval = FPToFixed(fltval, 0, FALSE, fpcr, rounding, intsize);  
X[d, intsize] = intval;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTNS (vector)

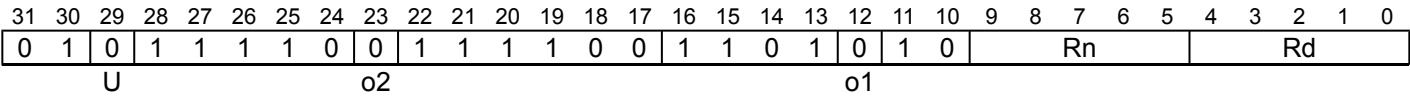
Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR EL1*, *CPTR EL2*, and *CPTR EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: *Scalar half precision* , *Scalar single-precision and double-precision* , *Vector half precision* and *Vector single-precision and double-precision*

Scalar half precision  
(FEAT\_FP16)



FCVTNS <Hd> , <Hn>

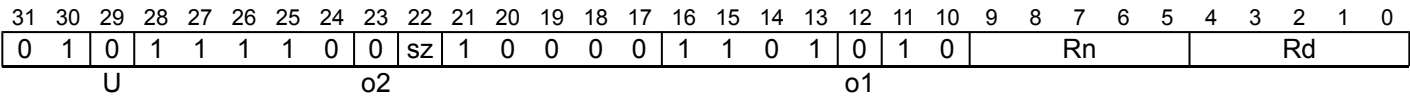
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision



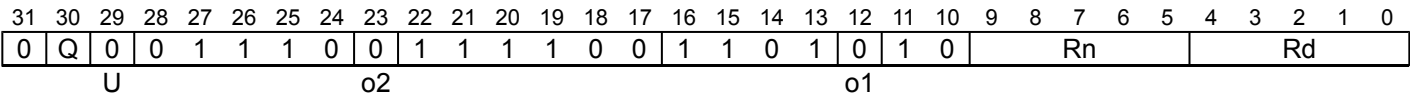
FCVTNS <V><d> , <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision  
(FEAT\_FP16)



FCVTNS <Vd>.<T>, <Vn>.<T>

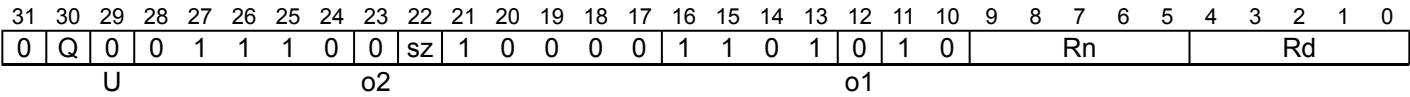
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTNS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();  
bits(datasize) operand = V[n, datasize];  
  
bits(esize) element;  
FPCRType fpcr = FPCR[];  
boolean merge = elements == 1 && IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
  
for e = 0 to elements-1  
    element = Elem[operand, e, esize];  
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTNU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		0	0	1	1	1	1	0	ftype		1	0	0	0	0	1	0	0	0	0	0	0	Rn				Rd				
rmode												opcode																			

Half-precision to 32-bit (sf == 0 && ftype == 11)  
(FEAT\_FP16)

```
FCVTNU <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)  
(FEAT\_FP16)

```
FCVTNU <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTNU <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTNU <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTNU <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTNU <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer decode_fltsize;
FPRounding rounding;

case ftype of
  when '00'
    decode_fltsize = 32;
  when '01'
    decode_fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      decode_fltsize = 16;
    else
      UNDEFINED;

rounding = FPDecodeRounding(rmode);
```



## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRTYPE fpcr = FPCR[];  
bits(decode_fltsize) fltval;  
bits(intsize) intval;  
  
fltval = V[n, decode_fltsize];  
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding, intsize);  
X[d, intsize] = intval;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTNU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

Scalar half precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn				Rd					
U								o2				o1																			

FCVTNU <Hd> , <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn				Rd					
U								o2				o1																			

FCVTNU <V><d> , <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn				Rd					
U								o2				o1																			

FCVTNU <Vd>.<T>, <Vn>.<T>

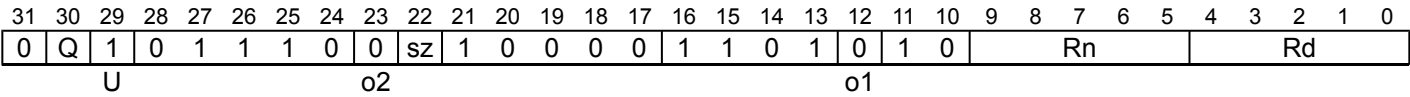
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTNU <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();  
bits(datasize) operand = V[n, datasize];  
  
bits(esize) element;  
FPCRType fpcr = FPCR[];  
boolean merge = elements == 1 && IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
  
for e = 0 to elements-1  
    element = Elem[operand, e, esize];  
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCVTPS (scalar)

Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	ftype	1	0	1	0	0	0	0	0	0	0	0	0	0	Rn				Rd					
rmode												opcode																			

### Half-precision to 32-bit (sf == 0 && ftype == 11) (FEAT\_FP16)

FCVTPS <Wd>, <Hn>

### Half-precision to 64-bit (sf == 1 && ftype == 11) (FEAT\_FP16)

FCVTPS <Xd>, <Hn>

### Single-precision to 32-bit (sf == 0 && ftype == 00)

FCVTPS <Wd>, <Sn>

### Single-precision to 64-bit (sf == 1 && ftype == 00)

FCVTPS <Xd>, <Sn>

### Double-precision to 32-bit (sf == 0 && ftype == 01)

FCVTPS <Wd>, <Dn>

### Double-precision to 64-bit (sf == 1 && ftype == 01)

FCVTPS <Xd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer decode_ftsize;
FPRounding rounding;

case ftype of
  when '00'
    decode_ftsize = 32;
  when '01'
    decode_ftsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      decode_ftsize = 16;
    else
      UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRTYPE fpcr = FPCR[];  
bits(decode_fltsize) fltval;  
bits(intsize) intval;  
  
fltval = V[n, decode_fltsize];  
intval = FPToFixed(fltval, 0, FALSE, fpcr, rounding, intsize);  
X[d, intsize] = intval;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTPS (vector)

Floating-point Convert to Signed integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

Scalar half precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn				Rd					
U				o2				o1																							

FCVTPS <Hd> , <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn				Rd					
U				o2				o1																							

FCVTPS <V><d> , <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn				Rd					
U				o2				o1																							

FCVTPS <Vd>.<T>, <Vn>.<T>

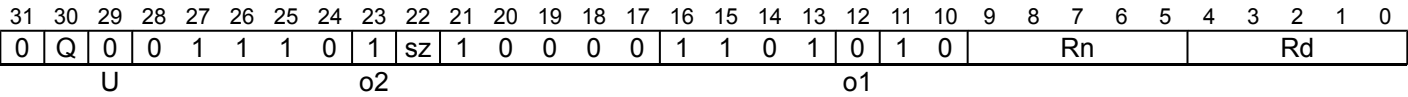
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTPS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.



## Operation

```
CheckFPAdvSIMDEnabled64();  
bits(datasize) operand = V[n, datasize];  
  
bits(esize) element;  
FPCRType fpcr = FPCR[];  
boolean merge = elements == 1 && IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
  
for e = 0 to elements-1  
    element = Elem[operand, e, esize];  
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

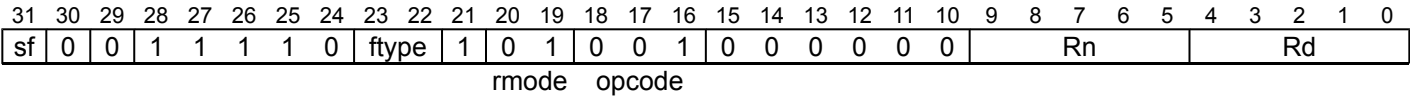
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTPU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision to 32-bit (sf == 0 && ftype == 11)  
(FEAT\_FP16)

```
FCVTPU <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)  
(FEAT\_FP16)

```
FCVTPU <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTPU <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTPU <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTPU <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTPU <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer decode_ftsize;
FPRounding rounding;

case ftype of
  when '00'
    decode_ftsize = 32;
  when '01'
    decode_ftsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      decode_ftsize = 16;
    else
      UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRTYPE fpcr = FPCR[];  
bits(decode_fltsize) fltval;  
bits(intsize) intval;  
  
fltval = V[n, decode_fltsize];  
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding, intsize);  
X[d, intsize] = intval;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCVTPU (vector)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#), [Scalar single-precision and double-precision](#), [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn				Rd					
U								o2				o1																			

**FCVTPU** <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	0	1	0	Rn				Rd					
U								o2				o1																			

**FCVTPU** <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Vector half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	0	1	0	Rn				Rd					
U								o2				o1																			

FCVTPU <Vd>.<T>, <Vn>.<T>

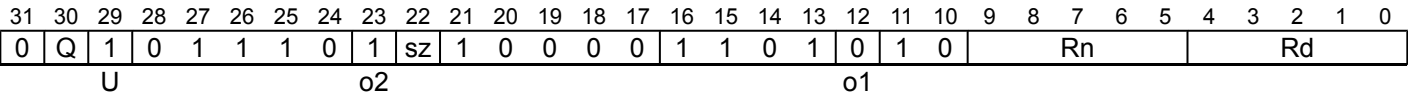
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTPU <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();  
bits(datasize) operand = V[n, datasize];  
  
bits(esize) element;  
FPCRType fpcr = FPCR[];  
boolean merge = elements == 1 && IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
  
for e = 0 to elements-1  
    element = Elem[operand, e, esize];  
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCVTXN, FCVTXN2

Floating-point Convert to lower precision Narrow, rounding to odd (vector). This instruction reads each vector element in the source SIMD&FP register, narrows each value to half the precision of the source element using the Round to Odd rounding mode, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

### Note

This instruction uses the Round to Odd rounding mode which is not defined by the IEEE 754-2008 standard. This rounding mode ensures that if the result of the conversion is inexact the least significant bit of the mantissa is forced to 1. This rounding mode enables a floating-point value to be converted to a lower precision format via an intermediate precision format while avoiding double rounding errors. For example, a 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value by first using this instruction to produce a 32-bit value and then using another instruction with the wanted rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

The FCVTXN instruction writes the vector to the lower half of the destination register and clears the upper half, while the FCVTXN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	0	1	1	0	1	0	Rn						Rd					

FCVTXN <Vb><d>, <Va><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '0' then UNDEFINED;
integer esize = 32;
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	0	1	1	0	1	0	Rn						Rd					

FCVTXN{2} <Vd>.<Tb>, <Vn>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '0' then UNDEFINED;
integer esize = 32;
integer datasize = 64;
integer elements = 2;
integer part = UInt(Q);
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<Tb>
0	x	RESERVED
1	0	2S
1	1	4S

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “sz”:

sz	<Ta>
0	RESERVED
1	2D

<Vb> Is the destination width specifier, encoded in “sz”:

sz	<Vb>
0	RESERVED
1	S

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in “sz”:

sz	<Va>
0	RESERVED
1	D

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(2*datasize) operand = V[n, 2*datasize];
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], fpcr, FPRounding_ODD, esize);

if merge then
    V[d, 128] = result;
else
    Vpart[d, part, datasize] = Elem[result, 0, datasize];
```



FCVTZS (scalar, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	ftype	0	1	1	0	0	0	scale					Rn					Rd						
rmode								opcode																							

Half-precision to 32-bit (sf == 0 && ftype == 11)  
(FEAT\_FP16)

```
FCVTZS <Wd>, <Hn>, #<fbits>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)  
(FEAT\_FP16)

```
FCVTZS <Xd>, <Hn>, #<fbits>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTZS <Wd>, <Sn>, #<fbits>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTZS <Xd>, <Sn>, #<fbits>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTZS <Wd>, <Dn>, #<fbits>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTZS <Xd>, <Dn>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer decode_fltsize;

case ftype of
  when '00' decode_fltsize = 32;
  when '01' decode_fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      decode_fltsize = 16;
    else
      UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<fbits>	For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".  For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

## Operation

```
CheckFPEnabled64();  
  
FPCRTType fpcr = FPCR[];  
bits(decode_fltsize) fltval;  
bits(intsize) intval;  
  
fltval = V[n, decode_fltsize];  
intval = FPToFixed(fltval, fracbits, FALSE, fpcr, FPRounding_ZERO, intsize);  
X[d, intsize] = intval;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

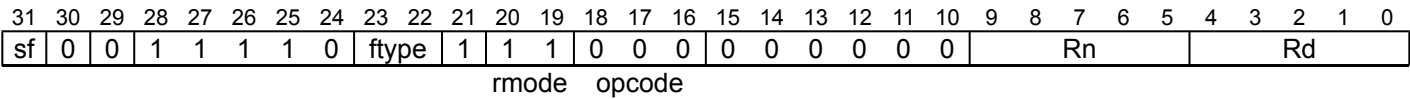
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTZS (scalar, integer)

Floating-point Convert to Signed integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision to 32-bit (sf == 0 && ftype == 11)  
(FEAT\_FP16)

```
FCVTZS <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)  
(FEAT\_FP16)

```
FCVTZS <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTZS <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTZS <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTZS <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTZS <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer decode_fltsize;
FPRounding rounding;

case ftype of
  when '00'
    decode_fltsize = 32;
  when '01'
    decode_fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      decode_fltsize = 16;
    else
      UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRTYPE fpcr = FPCR[];  
bits(decode_fltsize) fltval;  
bits(intsize) intval;  
  
fltval = V[n, decode_fltsize];  
intval = FPToFixed(fltval, 0, FALSE, fpcr, rounding, intsize);  
X[d, intsize] = intval;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCVTZS (vector, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR EL1](#), [CPTR EL2](#), and [CPTR EL3](#) registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			1	1	1	1	1	1	Rn			Rd						
U									immh																						

**FCVTZS** <V><d>, <V><n>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			1	1	1	1	1	1	Rn			Rd						
U									immh																						

**FCVTZS** <Vd>.<T>, <Vn>.<T>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

### Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
000x	RESERVED
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	x	RESERVED
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in "immh:immb":

immh	<fbits>
000x	RESERVED
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in "immh:immb":

immh	<fbits>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	RESERVED
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, fpcr, rounding, esize);

V[d, 128] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCVTZS (vector, integer)

Floating-point Convert to Signed integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#), [Scalar single-precision and double-precision](#), [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn				Rd					
U				o2				o1																							

**FCVTZS** <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn				Rd					
U				o2				o1																							

**FCVTZS** <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Vector half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn				Rd					
U				o2				o1																							

FCVTZS <Vd>.<T>, <Vn>.<T>

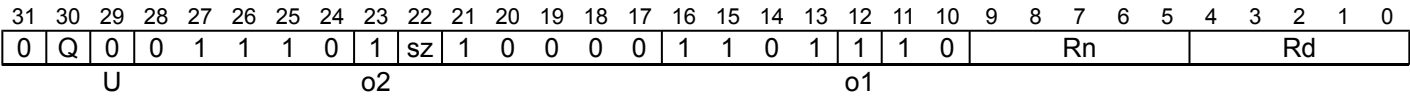
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTZS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.



## Operation

```
CheckFPAdvSIMDEnabled64();  
bits(datasize) operand = V[n, datasize];  
  
bits(esize) element;  
FPCRType fpcr = FPCR[];  
boolean merge = elements == 1 && IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
  
for e = 0 to elements-1  
    element = Elem[operand, e, esize];  
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

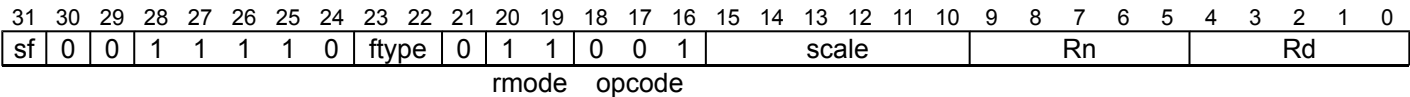
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTZU (scalar, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.



Half-precision to 32-bit (sf == 0 && ftype == 11)  
(FEAT\_FP16)

```
FCVTZU <Wd>, <Hn>, #<fbits>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)  
(FEAT\_FP16)

```
FCVTZU <Xd>, <Hn>, #<fbits>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTZU <Wd>, <Sn>, #<fbits>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTZU <Xd>, <Sn>, #<fbits>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTZU <Wd>, <Dn>, #<fbits>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTZU <Xd>, <Dn>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer decode_fltsize;

case ftype of
  when '00' decode_fltsize = 32;
  when '01' decode_fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      decode_fltsize = 16;
    else
      UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<fbits>	For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".  For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

## Operation

```
CheckFPEnabled64();  
  
FPCRTType fpcr = FPCR[];  
bits(decode_fltsize) fltval;  
bits(intsize) intval;  
  
fltval = V[n, decode_fltsize];  
intval = FPToFixed(fltval, fracbits, TRUE, fpcr, FPRounding_ZERO, intsize);  
X[d, intsize] = intval;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCVTZU (scalar, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	ftype	1	1	1	0	0	1	0	0	0	0	0	0	Rn					Rd					
rmode												opcode																			

### Half-precision to 32-bit (sf == 0 && ftype == 11) (FEAT\_FP16)

FCVTZU <Wd>, <Hn>

### Half-precision to 64-bit (sf == 1 && ftype == 11) (FEAT\_FP16)

FCVTZU <Xd>, <Hn>

### Single-precision to 32-bit (sf == 0 && ftype == 00)

FCVTZU <Wd>, <Sn>

### Single-precision to 64-bit (sf == 1 && ftype == 00)

FCVTZU <Xd>, <Sn>

### Double-precision to 32-bit (sf == 0 && ftype == 01)

FCVTZU <Wd>, <Dn>

### Double-precision to 64-bit (sf == 1 && ftype == 01)

FCVTZU <Xd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer decode_fltsize;
FPRounding rounding;

case ftype of
  when '00'
    decode_fltsize = 32;
  when '01'
    decode_fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      decode_fltsize = 16;
    else
      UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

## Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRTYPE fpcr = FPCR[];  
bits(decode_fltsize) fltval;  
bits(intsize) intval;  
  
fltval = V[n, decode_fltsize];  
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding, intsize);  
X[d, intsize] = intval;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCVTZU (vector, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR EL1](#), [CPTR EL2](#), and [CPTR EL3](#) registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0		1	1	1	1	1	1	0	!= 0000				immb			1	1	1	1	1	1	Rn			Rd						
U									immh																						

**FCVTZU** <V><d>, <V><n>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	1	1	1	1	1	Rn			Rd						
U									immh																						

**FCVTZU** <Vd>.<T>, <Vn>.<T>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

### Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
000x	RESERVED
001x	H
01xx	S
1xxx	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	x	RESERVED
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in "immh:immb":

immh	<fbits>
000x	RESERVED
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in "immh:immb":

immh	<fbits>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	RESERVED
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, fpcr, rounding, esize);

V[d, 128] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FCVTZU (vector, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision (FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn				Rd					
U								o2				o1																			

**FCVTZU** <Hd> , <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	0	1	1	1	0	Rn				Rd					
U								o2				o1																			

**FCVTZU** <V><d> , <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Vector half precision (FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	0	1	1	1	0	Rn				Rd					
U								o2				o1																			



FCVTZU <Vd>.<T>, <Vn>.<T>

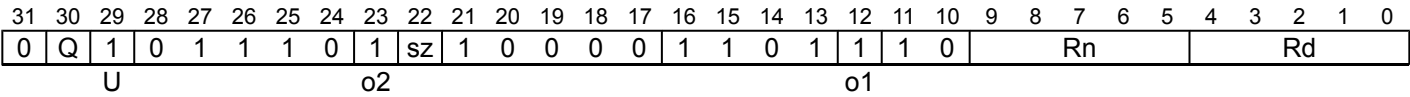
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTZU <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();  
bits(datasize) operand = V[n, datasize];  
  
bits(esize) element;  
FPCRType fpcr = FPCR[];  
boolean merge = elements == 1 && IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
  
for e = 0 to elements-1  
    element = Elem[operand, e, esize];  
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FDIV (scalar)

Floating-point Divide (scalar). This instruction divides the floating-point value of the first source SIMD&FP register by the floating-point value of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	ftype		1	Rm					0	0	0	1	1	0	Rn					Rd				

Half-precision (ftype == 11)  
(FEAT\_FP16)

FDIV <Hd>, <Hn>, <Hm>

Single-precision (ftype == 00)

FDIV <Sd>, <Sn>, <Sm>

Double-precision (ftype == 01)

FDIV <Dd>, <Dn>, <Dm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPEEnabled64();  
bits(esize) operand1 = V[n, esize];  
bits(esize) operand2 = V[m, esize];  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[n, 128] else Zeros(128);  
  
Elem[result, 0, esize] = FPDiv(operand1, operand2, FPCR[]);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FDIV (vector)

Floating-point Divide (vector). This instruction divides the floating-point values in the elements in the first source SIMD&FP register, by the floating-point values in the corresponding elements in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Half-precision* and *Single-precision and double-precision*

Half-precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	0	Rm				0	0	1	1	1	1	Rn				Rd						

FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm				1	1	1	1	1	1	Rn				Rd						

FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>

For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPDIV(element1, element2, FPCR[]);

V[d, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FJCVTZS

Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero. This instruction converts the double-precision floating-point value in the SIMD&FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register. If the result is too large to be represented as a signed 32-bit integer, then the result is the integer modulo 2<sup>32</sup>, as held in a 32-bit signed integer. This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Double-precision to 32-bit  
(FEAT\_JSCVT)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	Rn				Rd					
sf				ftype				rmode				opcode																			

FJCVTZS <Wd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HaveFJCVTZSExt() then UNDEFINED;
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
FPCRType fpcr = FPCR[];
bits(64) fltval;
bits(32) intval;

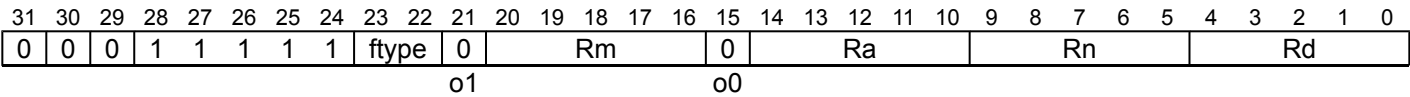
bit z;
fltval = V[n, 64];
(intval, z) = FPToFixedJS(fltval, fpcr, TRUE, 32);
PSTATE.<N,Z,C,V> = '0':z:'00';
X[d, 32] = intval;
```

FMADD

Floating-point fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, adds the product to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

```
FMADD <Hd>, <Hn>, <Hm>, <Ha>
```

Single-precision (ftype == 00)

```
FMADD <Sd>, <Sn>, <Sm>, <Sa>
```

Double-precision (ftype == 01)

```
FMADD <Dd>, <Dn>, <Dm>, <Da>
```

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- <Da> Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- <Ha> Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.



<Sn>	Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Sa>	Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

## Operation

```

CheckFPEnabled64();

bits(esize) operanda = V[a, esize];
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a, 128] else Zeros(128);

Elem[result, 0, esize] = FPMulAdd(operanda, operand1, operand2, fpcr);

V[d, 128] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAX (scalar)

Floating-point Maximum (scalar). This instruction compares the two source SIMD&FP registers, and writes the larger of the two floating-point values to the destination SIMD&FP register.

When *FPCR*.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When *FPCR*.DN is 0, if either value is a NaN, the result is a quiet NaN.
- When *FPCR*.DN is 1, if either value is a NaN, the result is Default NaN.

When *FPCR*.AH is 1, the behavior is as follows:

- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of *FPCR*.DN, the result is the second value.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	ftype	1	Rm				0	1	0	0	1	0	Rn				Rd							
op																															

Half-precision (ftype == 11)  
(FEAT\_FP16)

```
FMAX <Hd>, <Hn>, <Hm>
```

Single-precision (ftype == 00)

```
FMAX <Sd>, <Sn>, <Sm>
```

Double-precision (ftype == 01)

```
FMAX <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPEnabled64();  
bits(esize) operand1 = V[n, esize];  
bits(esize) operand2 = V[m, esize];  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[n, 128] else Zeros(128);  
  
Elem[result, 0, esize] = FPMMax(operand1, operand2, fpcr);  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMAX (vector)

Floating-point Maximum (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the larger of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

When [FPCR.AH](#) is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When [FPCR.DN](#) is 0, if either element is a NaN, the result is a quiet NaN.
- When [FPCR.DN](#) is 1, if either element is a NaN, the result is Default NaN.

When [FPCR.AH](#) is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of [FPCR.DN](#), the result is the second element.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	0	Rm				0	0	1	1	0	1	Rn				Rd						
U								o1																							

**FMAX** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

```
boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	Rm				1	1	1	1	0	1	Rn				Rd						
U								o1																							

**FMAX** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

```
boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];

  if minimum then
    Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);

V[d, datasize] = result;
```

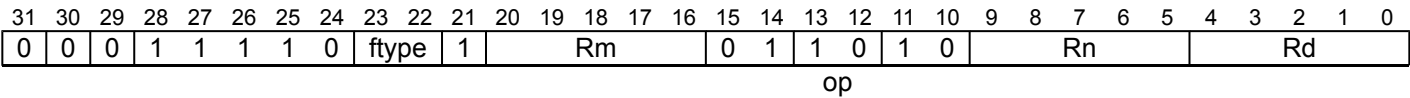
FMAXNM (scalar)

Floating-point Maximum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the larger of the two floating-point values to the destination SIMD&FP register. Regardless of the value of *FPCR.AH*, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When *FPCR.DN* is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When *FPCR.DN* is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

```
FMAXNM <Hd>, <Hn>, <Hm>
```

Single-precision (ftype == 00)

```
FMAXNM <Sd>, <Sn>, <Sm>
```

Double-precision (ftype == 01)

```
FMAXNM <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPEnabled64();  
bits(esize) operand1 = V[n, esize];  
bits(esize) operand2 = V[m, esize];  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[n, 128] else Zeros(128);  
  
Elem[result, 0, esize] = FPMaxNum(operand1, operand2, fpcr);  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMAXNM (vector)

Floating-point Maximum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the larger of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

Regardless of the value of [FPCR.AH](#), the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When [FPCR.DN](#) is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When [FPCR.DN](#) is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	0	Rm				0	0	0	0	0	1	Rn				Rd						
U								a																							

**FMAXNM** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	Rm				1	1	0	0	0	1	Rn				Rd						
U								o1																							

**FMAXNM** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```



Assembler Symbols

- <Vd>Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn>Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm>Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];

  if minimum then
    Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);

V[d, datasize] = result;
```

FMAXNMP (scalar)

Floating-point Maximum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

Regardless of the value of `FPCR.AH`, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When `FPCR.DN` is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When `FPCR.DN` is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in `FPCR`, the exception results in either a flag being set in `FPSR` or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
o1																															

FMAXNMP <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
o1																															

FMAXNMP <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_FMAXNUM, operand, esize, FALSE);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMAXNMP (vector)

Floating-point Maximum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

Regardless of the value of [FPCR.AH](#), the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When [FPCR.DN](#) is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When [FPCR.DN](#) is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	0	Rm				0	0	0	0	0	1	Rn				Rd						
U								a																							

**FMAXNMP** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm				1	1	0	0	0	1	Rn				Rd						
U								o1																							

**FMAXNMP** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

- <Vd>Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn>Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm>Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];

  if minimum then
    Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);

V[d, datasize] = result;
```

FMAXNMV

Floating-point Maximum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

Regardless of the value of `FPCR.AH`, the behavior is as follows:

- Negative zero compares less than positive zero.
  - If one value is numeric and the other is a quiet NaN, the result is the numeric value.
  - When `FPCR.DN` is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
  - When `FPCR.DN` is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.
- This instruction can generate a floating-point exception. Depending on the settings in `FPCR`, the exception results in either a flag being set in `FPSR` or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).
- Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
- It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
o1																															

```
FMAXNMV <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
o1																															

```
FMAXNMV <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;    // .4S only

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

- <V> For the half-precision variant: is the destination width specifier, H.
- For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:
- | sz | <V>      |
|----|----------|
| 0  | S        |
| 1  | RESERVED |
- <d> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_FMAXNUM, operand, esize, FALSE);
```

FMAXP (scalar)

Floating-point Maximum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register. When `FPCR.AH` is 0, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- When `FPCR.DN` is 0, if either element is a NaN, the result is a quiet NaN.
- When `FPCR.DN` is 1, if either element is a NaN, the result is Default NaN.

When `FPCR.AH` is 1, the behavior is as follows for each pairwise operation:

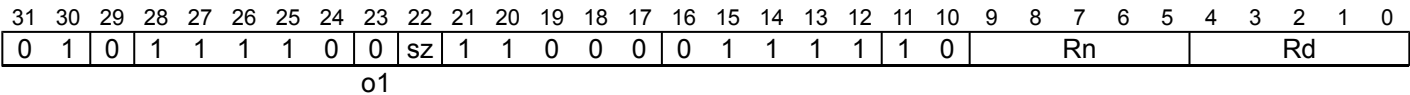
- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of `FPCR.DN`, the result is the second element.

This instruction can generate a floating-point exception. Depending on the settings in `FPCR`, the exception results in either a flag being set in `FPSR` or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(FEAT\_FP16)



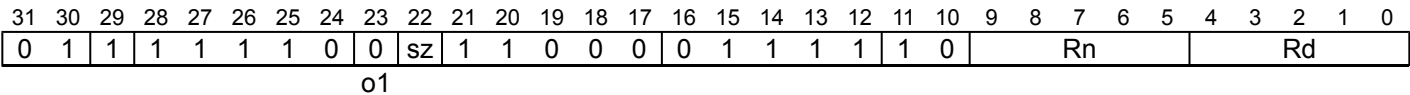
```
FMAXP <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;
```

Single-precision and double-precision



```
FMAXP <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:



<b>sz</b>	<b>&lt;V&gt;</b>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in "sz":

<b>sz</b>	<b>&lt;T&gt;</b>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in "sz":

<b>sz</b>	<b>&lt;T&gt;</b>
0	2S
1	2D

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

V[d, esize] = Reduce(ReduceOp_FMAX, operand, esize);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMAXP (vector)

Floating-point Maximum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the larger of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

When [FPCR.AH](#) is 0, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- When [FPCR.DN](#) is 0, if either element is a NaN, the result is a quiet NaN.
- When [FPCR.DN](#) is 1, if either element is a NaN, the result is Default NaN.

When [FPCR.AH](#) is 1, the behavior is as follows for each pairwise operation:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of [FPCR.DN](#), the result is the second element.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	0	Rm				0	0	1	1	0	1	Rn				Rd						
U								o1																							

**FMAXP** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm				1	1	1	1	0	1	Rn				Rd						
U								o1																							

**FMAXP** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];

  if minimum then
    Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);

V[d, datasize] = result;
```

FMAXV

Floating-point Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

When `FPCR.AH` is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When `FPCR.DN` is 0, if either value is a NaN, the result is a quiet NaN.
- When `FPCR.DN` is 1, if either value is a NaN, the result is Default NaN.

When `FPCR.AH` is 1, the behavior is as follows:

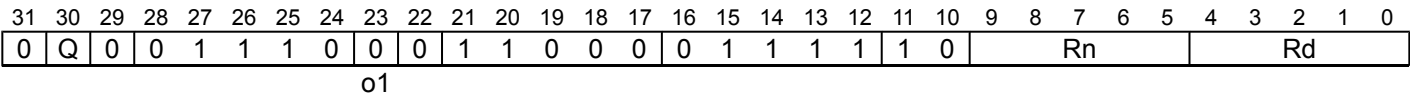
- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of `FPCR.DN`, the result is the second value.

This instruction can generate a floating-point exception. Depending on the settings in `FPCR`, the exception results in either a flag being set in `FPSR` or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(FEAT\_FP16)



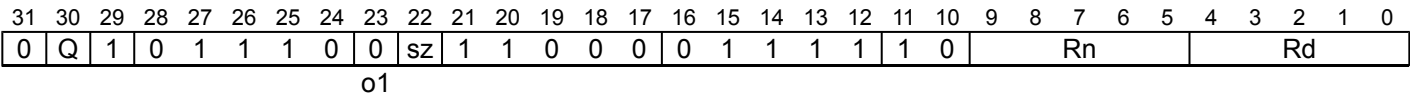
FMAXV <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

Single-precision and double-precision



FMAXV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

- <V> For the half-precision variant: is the destination width specifier, H.
- For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

<b>sz</b>	<b>&lt;V&gt;</b>
0	S
1	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<b>Q</b>	<b>&lt;T&gt;</b>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<b>Q</b>	<b>sz</b>	<b>&lt;T&gt;</b>
0	x	RESERVED
1	0	4S
1	1	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_FMAX, operand, esize);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMIN (scalar)

Floating-point Minimum (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

When *FPCR.AH* is 0, the behavior is as follows:

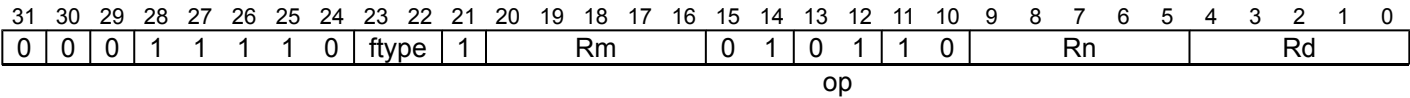
- Negative zero compares less than positive zero.
- When *FPCR.DN* is 0, if either value is a NaN, the result is a quiet NaN.
- When *FPCR.DN* is 1, if either value is a NaN, the result is Default NaN.

When *FPCR.AH* is 1, the behavior is as follows:

- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of *FPCR.DN*, the result is the second value.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

```
FMIN <Hd>, <Hn>, <Hm>
```

Single-precision (ftype == 00)

```
FMIN <Sd>, <Sn>, <Sm>
```

Double-precision (ftype == 01)

```
FMIN <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPEnabled64();  
bits(esize) operand1 = V[n, esize];  
bits(esize) operand2 = V[m, esize];  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[n, 128] else Zeros(128);  
  
Elem[result, 0, esize] = FPMIn(operand1, operand2, fpcr);  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMIN (vector)

Floating-point minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

When **FPCR.AH** is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When **FPCR.DN** is 0, if either element is a NaN, the result is a quiet NaN.
- When **FPCR.DN** is 1, if either element is a NaN, the result is Default NaN.

When **FPCR.AH** is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of **FPCR.DN**, the result is the second element.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the **CPACR\_EL1**, **CPTR\_EL2**, and **CPTR\_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision (FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	0	Rm				0	0	1	1	0	1	Rn				Rd						
U								o1																							

**FMIN** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	Rm				1	1	1	1	0	1	Rn				Rd						
U								o1																							

**FMIN** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```



Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];

  if minimum then
    Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);

V[d, datasize] = result;
```

## FMINNM (scalar)

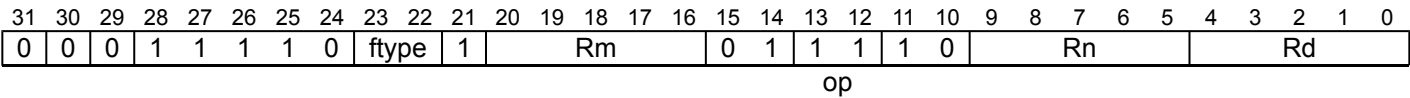
Floating-point Minimum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

Regardless of the value of `FPCR.AH`, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When `FPCR.DN` is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When `FPCR.DN` is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in `FPCR`, the exception results in either a flag being set in `FPSR` or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### Half-precision (ftype == 11) (FEAT\_FP16)

FMINNM <Hd>, <Hn>, <Hm>

### Single-precision (ftype == 00)

FMINNM <Sd>, <Sn>, <Sm>

### Double-precision (ftype == 01)

FMINNM <Dd>, <Dn>, <Dm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

## Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPEnabled64();  
bits(esize) operand1 = V[n, esize];  
bits(esize) operand2 = V[m, esize];  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[n, 128] else Zeros(128);  
  
Elem[result, 0, esize] = FPMinNum(operand1, operand2, fpcr);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMINNM (vector)

Floating-point Minimum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the smaller of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

Regardless of the value of [FPCR.AH](#), the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When [FPCR.DN](#) is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When [FPCR.DN](#) is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	0	Rm				0	0	0	0	0	1	Rn				Rd						
U								a																							

**FMINNM** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	sz	1	Rm				1	1	0	0	0	1	Rn				Rd						
U								o1																							

**FMINNM** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

- <Vd>Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn>Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm>Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];

  if minimum then
    Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);

V[d, datasize] = result;
```

FMINNMP (scalar)

Floating-point Minimum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

Regardless of the value of *FPCR.AH*, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When *FPCR.DN* is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When *FPCR.DN* is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
o1																															

FMINNMP <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
o1																															

FMINNMP <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_FMINNUM, operand, esize, FALSE);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMINNMP (vector)

Floating-point Minimum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of floating-point values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

Regardless of the value of [FPCR.AH](#), the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When [FPCR.DN](#) is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When [FPCR.DN](#) is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	0	Rm				0	0	0	0	0	1	Rn				Rd						
U								a																							

**FMINNMP** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	Rm				1	1	0	0	0	1	Rn				Rd						
U								o1																							

**FMINNMP** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```



Assembler Symbols

- <Vd>Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn>Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm>Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];

  if minimum then
    Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);

V[d, datasize] = result;
```

FMINNMV

Floating-point Minimum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

Regardless of the value of `FPCR.AH`, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When `FPCR.DN` is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When `FPCR.DN` is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in `FPCR`, the exception results in either a flag being set in `FPSR` or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	0	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
o1																															

```
FMINNMV <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	0	0	1	0	Rn				Rd					
o1																															

```
FMINNMV <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;    // .4S only

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

- <V>

For the half-precision variant: is the destination width specifier, H.
- For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:
- | sz | <V>      |
|----|----------|
| 0  | S        |
| 1  | RESERVED |
- <d>

Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_FMINNUM, operand, esize, FALSE);
```

FMINP (scalar)

Floating-point Minimum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register. When `FPCR.AH` is 0, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- When `FPCR.DN` is 0, if either element is a NaN, the result is a quiet NaN.
- When `FPCR.DN` is 1, if either element is a NaN, the result is Default NaN.

When `FPCR.AH` is 1, the behavior is as follows for each pairwise operation:

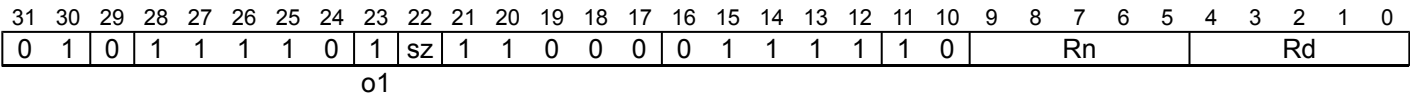
- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of `FPCR.DN`, the result is the second element.

This instruction can generate a floating-point exception. Depending on the settings in `FPCR`, the exception results in either a flag being set in `FPSR` or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(FEAT\_FP16)



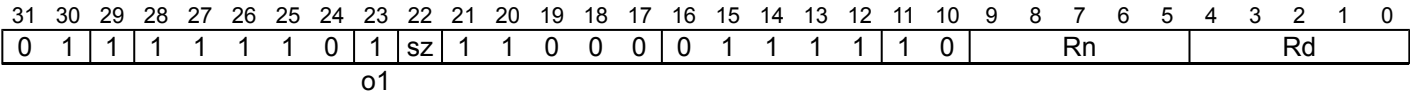
FMINP <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;
```

Single-precision and double-precision



FMINP <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in "sz":

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in "sz":

sz	<T>
0	2S
1	2D

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_FMIN, operand, esize);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMINP (vector)

Floating-point Minimum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the smaller of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

When [FPCR.AH](#) is 0, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- When [FPCR.DN](#) is 0, if either element is a NaN, the result is a quiet NaN.
- When [FPCR.DN](#) is 1, if either element is a NaN, the result is Default NaN.

When [FPCR.AH](#) is 1, the behavior is as follows for each pairwise operation:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of [FPCR.DN](#), the result is the second element.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	0	Rm				0	0	1	1	0	1	Rn				Rd						
U								o1																							

**FMINP** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	Rm				1	1	1	1	0	1	Rn				Rd						
U								o1																							

**FMINP** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];

  if minimum then
    Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);

V[d, datasize] = result;
```

FMINV

Floating-point Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

When **FPCR.AH** is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When **FPCR.DN** is 0, if either value is a NaN, the result is a quiet NaN.
- When **FPCR.DN** is 1, if either value is a NaN, the result is Default NaN.

When **FPCR.AH** is 1, the behavior is as follows:

- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of **FPCR.DN**, the result is the second value.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR** or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the **CPACR\_EL1**, **CPTR\_EL2**, and **CPTR\_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
o1																															

FMINV <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn				Rd					
o1																															

FMINV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

- <V> For the half-precision variant: is the destination width specifier, H.
- For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:



<b>sz</b>	<b>&lt;V&gt;</b>
0	S
1	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<b>Q</b>	<b>&lt;T&gt;</b>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<b>Q</b>	<b>sz</b>	<b>&lt;T&gt;</b>
0	x	RESERVED
1	0	4S
1	1	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_FMIN, operand, esize);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMLA (by element)

Floating-point fused Multiply-Add to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results in the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: *Scalar, half-precision* , *Scalar, single-precision and double-precision* , *Vector, half-precision* and *Vector, single-precision and double-precision*

Scalar, half-precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	0	L	M	Rm				0	0	0	1	H	0	Rn				Rd					
o2																															

FMLA <Hd>, <Hn>, <Vm>.H[<index>]

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

Scalar, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	1	sz	L	M	Rm				0	0	0	1	H	0	Rn				Rd					
o2																															

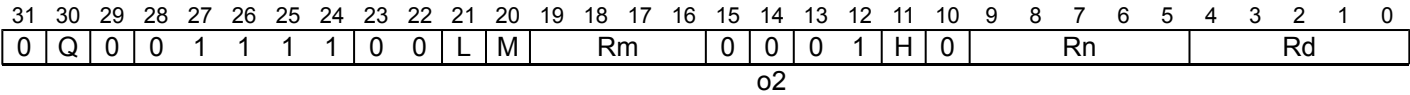
FMLA <V><d>, <V><n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

Vector, half-precision  
(FEAT\_FP16)



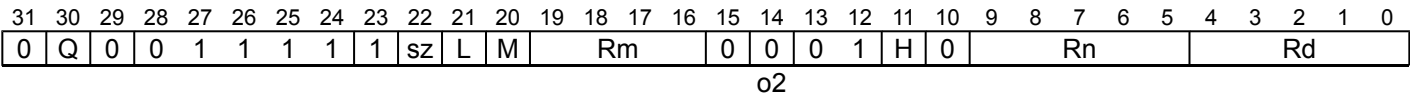
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Vector, single-precision and double-precision



FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Assembler Symbols

- <Hd>

Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn>

Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <V>

Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D
- <d>

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n>

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in "sz":

sz	<Ts>
0	S
1	D

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in "sz:L:H":

sz	L	<index>
0	x	H:L
1	0	H
1	1	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) operand3 = V[d, datasize];
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, fpcr);

V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMLA (vector)

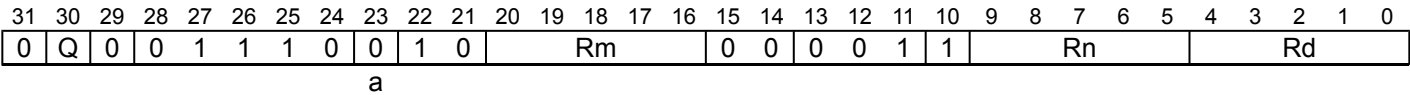
Floating-point fused Multiply-Add to accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, adds the product to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR EL1*, *CPTR EL2*, and *CPTR EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Half-precision* and *Single-precision and double-precision*

Half-precision  
(FEAT\_FP16)



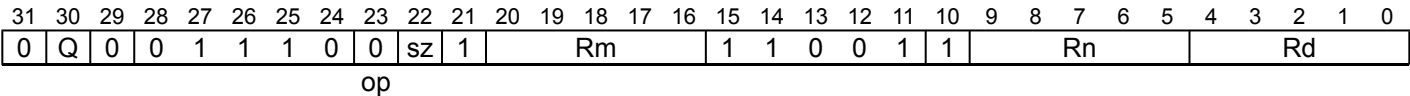
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (a == '1');
```

Single-precision and double-precision



FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (op == '1');
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMLAL, FMLAL2 (by element)

Floating-point fused Multiply-Add Long to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

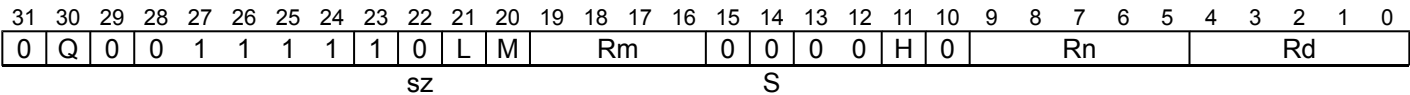
In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

*ID\_AA64ISAR0\_EL1*.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: *FMLAL* and *FMLAL2*

FMLAL  
(FEAT\_FHM)



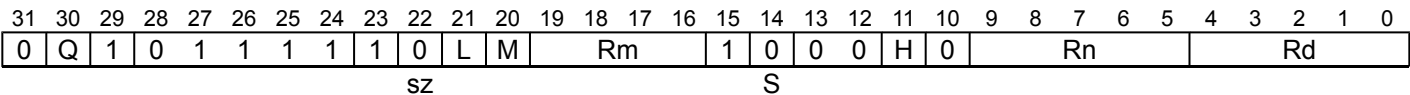
```
FMLAL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<H>[<index>]

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 0;
```

FMLAL2  
(FEAT\_FHM)



FMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<H>[<index>]

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 1;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "Q":

Q	<Tb>
0	2H
1	4H

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<index> Is the element index, encoded in the "H:L:M" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part, datasize DIV 2];
bits(128) operand2 = V[m, 128];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d, datasize] = result;
```



## FMLAL, FMLAL2 (vector)

Floating-point fused Multiply-Add Long to accumulator (vector). This instruction multiplies corresponding half-precision floating-point values in the vectors in the two source SIMD&FP registers, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

### Note

*ID\_AA64ISAR0\_EL1*.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: *FMLAL* and *FMLAL2*

### FMLAL (FEAT\_FHM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	1	Rm				1	1	1	0	1	1	Rn				Rd						
S											sz																				

**FMLAL** <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 0;
```

### FMLAL2 (FEAT\_FHM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	0	1	Rm				1	1	0	0	1	1	Rn				Rd						
S											sz																				

**FMLAL2** <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 1;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	2H
1	4H

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part, datasize DIV 2];
bits(datasize DIV 2) operand2 = Vpart[m, part, datasize DIV 2];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    element2 = Elem[operand2, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d, datasize] = result;
```

FMLS (by element)

Floating-point fused Multiply-Subtract from accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

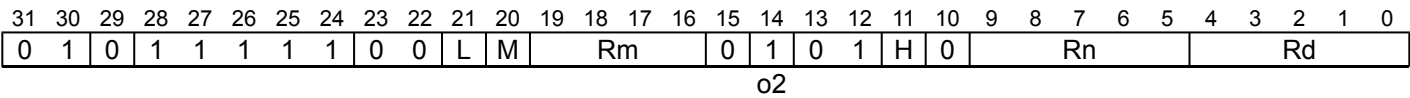
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: *Scalar, half-precision* , *Scalar, single-precision and double-precision* , *Vector, half-precision* and *Vector, single-precision and double-precision*

Scalar, half-precision

(FEAT\_FP16)



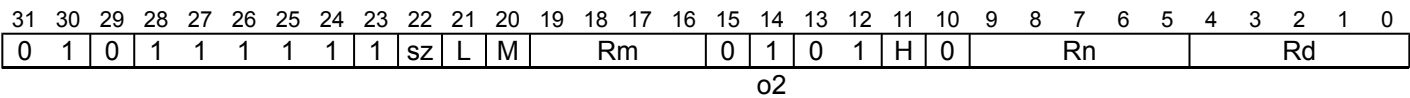
FMLS <Hd>, <Hn>, <Vm>.H[<index>]

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

Scalar, single-precision and double-precision



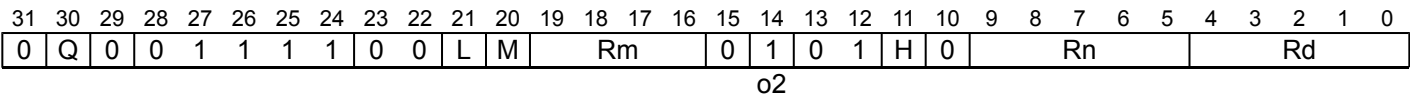
FMLS <V><d>, <V><n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

Vector, half-precision  
(FEAT\_FP16)



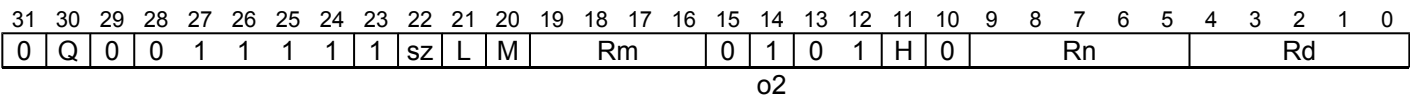
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<H>[<index>]

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Vector, single-precision and double-precision



FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D
- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in "sz":

sz	<Ts>
0	S
1	D

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in "sz:L:H":

sz	L	<index>
0	x	H:L
1	0	H
1	1	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) operand3 = V[d, datasize];
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, fpcr);

V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMLS (vector)

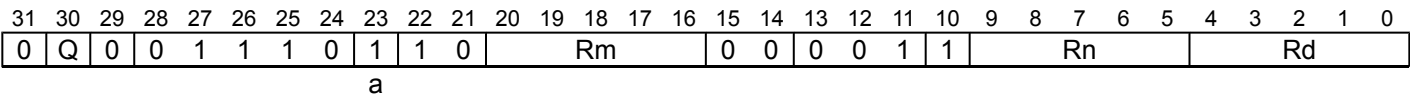
Floating-point fused Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, negates the product, adds the result to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Half-precision* and *Single-precision and double-precision*

Half-precision  
(FEAT\_FP16)



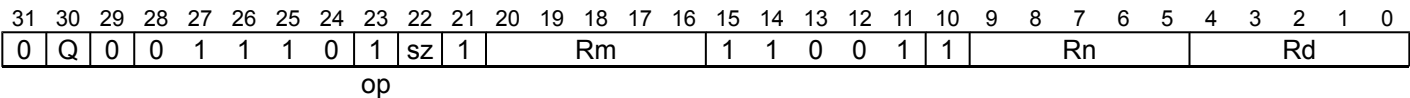
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (a == '1');
```

Single-precision and double-precision



FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (op == '1');
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMLSL, FMLSL2 (by element)

Floating-point fused Multiply-Subtract Long from accumulator (by element). This instruction multiplies the negated vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

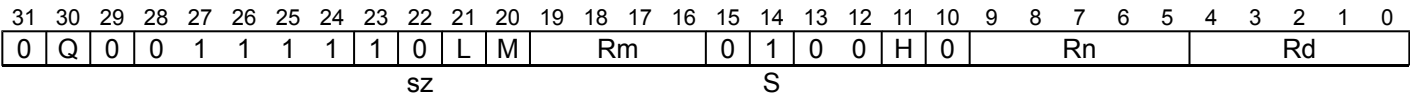
In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

*ID\_AA64ISAR0\_EL1*.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: *FMLSL* and *FMLSL2*

FMLSL  
(FEAT\_FHM)



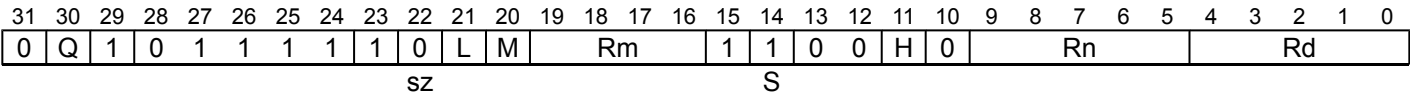
FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<H>[<index>]

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 0;
```

FMLSL2  
(FEAT\_FHM)





**FMLSL2** <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.&b>H[<index>]

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm);    // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 1;
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "Q":

Q	<Tb>
0	2H
1	4H

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<index> Is the element index, encoded in the "H:L:M" fields.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part, datasize DIV 2];
bits(128) operand2 = V[m, 128];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d, datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMLSL, FMLSL2 (vector)

Floating-point fused Multiply-Subtract Long from accumulator (vector). This instruction negates the values in the vector of one SIMD&FP register, multiplies these with the corresponding values in another vector, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

### Note

*ID\_AA64ISAR0\_EL1*.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: *FMLSL* and *FMLSL2*

### FMLSL (FEAT\_FHM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	0	1	Rm				1	1	1	0	1	1	Rn				Rd						
S sz																															

**FMLSL** <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 0;
```

### FMLSL2 (FEAT\_FHM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	0	1	Rm				1	1	0	0	1	1	Rn				Rd						
S sz																															

**FMLSL2** <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 1;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	2H
1	4H

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part, datasize DIV 2];
bits(datasize DIV 2) operand2 = Vpart[m, part, datasize DIV 2];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    element2 = Elem[operand2, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d, datasize] = result;
```

FMOV (general)

Floating-point Move to or from general-purpose register without conversion. This instruction transfers the contents of a SIMD&FP register to a general-purpose register, or the contents of a general-purpose register to a SIMD&FP register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
sf	0	0	1	1	1	1	0	f	t	y	p	e	1	0	x	1	1	x	0	0	0	0	0	0	Rn				Rd			
rmode												opcode																				

**Half-precision to 32-bit (sf == 0 && ftype == 11 && rmode == 00 && opcode == 110)**  
(FEAT\_FP16)

FMOV <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && ftype == 11 && rmode == 00 && opcode == 110)**  
(FEAT\_FP16)

FMOV <Xd>, <Hn>

**32-bit to half-precision (sf == 0 && ftype == 11 && rmode == 00 && opcode == 111)**  
(FEAT\_FP16)

FMOV <Hd>, <Wn>

**32-bit to single-precision (sf == 0 && ftype == 00 && rmode == 00 && opcode == 111)**

FMOV <Sd>, <Wn>

**Single-precision to 32-bit (sf == 0 && ftype == 00 && rmode == 00 && opcode == 110)**

FMOV <Wd>, <Sn>

**64-bit to half-precision (sf == 1 && ftype == 11 && rmode == 00 && opcode == 111)**  
(FEAT\_FP16)

FMOV <Hd>, <Xn>

**64-bit to double-precision (sf == 1 && ftype == 01 && rmode == 00 && opcode == 111)**

FMOV <Dd>, <Xn>

**64-bit to top half of 128-bit (sf == 1 && ftype == 10 && rmode == 01 && opcode == 111)**

FMOV <Vd>.D[1], <Xn>

**Double-precision to 64-bit (sf == 1 && ftype == 01 && rmode == 00 && opcode == 110)**

FMOV <Xd>, <Dn>

**Top half of 128-bit to 64-bit (sf == 1 && ftype == 10 && rmode == 01 && opcode == 110)**

FMOV <Xd>, <Vn>.D[1]

```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer decode_fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case ftype of
  when '00'
    decode_fltsize = 32;
  when '01'
    decode_fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '11 01' then UNDEFINED;
    decode_fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      decode_fltsize = 16;
    else
      UNDEFINED;

case opcode<2:1>:rmode of
  when '00 xx' // FCVT[NPMZ][US]
    rounding = FPDecodeRounding(rmode);
    unsigned = (opcode<0> == '1');
    op = FPConvOp\_CVT\_FtoI;
  when '01 00' // [US]CVTF
    rounding = FPRoundingMode(FPCR[]);
    unsigned = (opcode<0> == '1');
    op = FPConvOp\_CVT\_ItoF;
  when '10 00' // FCVTA[US]
    rounding = FPRounding\_TIEAWAY;
    unsigned = (opcode<0> == '1');
    op = FPConvOp\_CVT\_FtoI;
  when '11 00' // FMOV
    if decode_fltsize != 16 && decode_fltsize != intsize then UNDEFINED;
    op = if opcode<0> == '1' then FPConvOp\_MOV\_ItoF else FPConvOp\_MOV\_FtoI;
    part = 0;
  when '11 01' // FMOV D[1]
    if intsize != 64 || decode_fltsize != 128 then UNDEFINED;
    op = if opcode<0> == '1' then FPConvOp\_MOV\_ItoF else FPConvOp\_MOV\_FtoI;
    part = 1;
    decode_fltsize = 64; // size of D[1] is 64
  when '11 11' // FJCVTZS
    if !HaveFJCVTZSExt() then UNDEFINED;
    rounding = FPRounding\_ZERO;
    unsigned = (opcode<0> == '1');
    op = FPConvOp\_CVT\_FtoI\_JS;
  otherwise
    UNDEFINED;

```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Vn>	Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

if op == FPConvOp\_CVT\_FtoI\_JS then
    CheckFPAdvSIMDEnabled64();
else
    CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
constant integer fltsize = if op == FPConvOp\_CVT\_ItoF && merge then 128 else decode_fltsize;
bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp\_CVT\_FtoI
        fltval = V[n, fltsize];
        intval = FPToFixed(fltval, 0, unsigned, fpcr, rounding, intsize);
        X[d, intsize] = intval;
    when FPConvOp\_CVT\_ItoF
        intval = X[n, intsize];
        fltval = if merge then V[d, fltsize] else Zeros(fltsize);
        Elem[fltval, 0, decode_fltsize] = FixedToFP(intval, 0, unsigned, fpcr, rounding, decode_fltsize);
        V[d, fltsize] = fltval;
    when FPConvOp\_MOV\_FtoI
        fltval = Vpart[n, part, fltsize];
        intval = ZeroExtend(fltval, intsize);
        X[d, intsize] = intval;
    when FPConvOp\_MOV\_ItoF
        intval = X[n, intsize];
        fltval = intval<fltsize-1:0>;
        Vpart[d, part, fltsize] = fltval;
    when FPConvOp\_CVT\_FtoI\_JS
        bit z;
        fltval = V[n, fltsize];
        (intval, z) = FPToFixedJS(fltval, fpcr, TRUE, intsize);
        PSTATE.<N,Z,C,V> = '0':z:'00';
        X[d, intsize] = intval;

```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

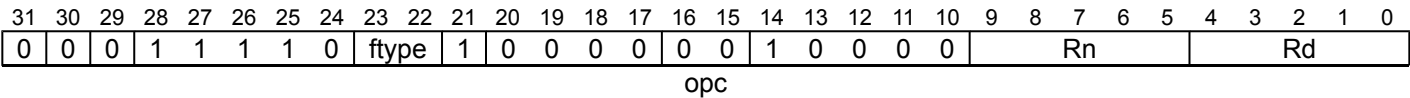
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMOV (register)

Floating-point Move register without conversion. This instruction copies the floating-point value in the SIMD&FP source register to the SIMD&FP destination register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

```
FMOV <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FMOV <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FMOV <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEEnabled64();

bits(128) result = 0<127:0>;

bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = operand;
V[d, 128] = result;
```





## FMOV (scalar, immediate)

Floating-point move immediate (scalar). This instruction copies a floating-point immediate constant into the SIMD&FP destination register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	ftype	1	imm8								1	0	0	0	0	0	0	Rd						

### Half-precision (ftype == 11) (FEAT\_FP16)

FMOV <Hd>, #<imm>

### Single-precision (ftype == 00)

FMOV <Sd>, #<imm>

### Double-precision (ftype == 01)

FMOV <Dd>, #<imm>

```
integer d = UInt(Rd);
integer datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UNDEFINED;
bits(datasize) imm = VFPEExpandImm(imm8, datasize);
```

## Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <imm> Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in the "imm8" field. For details of the range of constants available and the encoding of <imm>, see [Modified immediate constants in A64 floating-point instructions](#).

## Operation

```
CheckFPEnabled64();
V[d, datasize] = imm;
```

## FMOV (vector, immediate)

Floating-point move immediate (vector). This instruction copies an immediate floating-point constant into every element of the SIMD&FP destination register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	0	0	0	0	a	b	c	1	1	1	1	1	1	d	e	f	g	h	Rd				

**FMOV** <Vd>.<T>, #<imm>

```
if !HaveFP16Ext() then UNDEFINED;

integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;

bits(8) imm8 = a:b:c:d:e:f:g:h;
bits(16) imm16 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 2):imm8<5:0>:Zeros(6);

imm = Replicate(imm16, datasize DIV 16);
```

### Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	op	0	1	1	1	1	0	0	0	0	0	a	b	c	1	1	1	1	0	1	d	e	f	g	h	Rd				

cmode

#### Single-precision (op == 0)

**FMOV** <Vd>.<T>, #<imm>

#### Double-precision (Q == 1 && op == 1)

**FMOV** <Vd>.2D, #<imm>

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

if cmode:op == '11111' then
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm> Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in "a:b:c:d:e:f:g:h". For details of the range of constants available and the encoding of <imm>, see [Modified immediate constants in A64 floating-point instructions](#).

## Operation

```
CheckFPAdvSIMDEnabled64();
```

```
V[rd, datasize] = imm;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

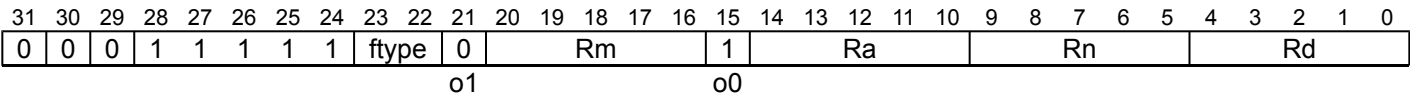
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# FMSUB

Floating-point Fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, adds that to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



## Half-precision (ftype == 11) (FEAT\_FP16)

```
FMSUB <Hd>, <Hn>, <Hm>, <Ha>
```

## Single-precision (ftype == 00)

```
FMSUB <Sd>, <Sn>, <Sm>, <Sa>
```

## Double-precision (ftype == 01)

```
FMSUB <Dd>, <Dn>, <Dm>, <Da>
```

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

## Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- <Da> Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Ha>	Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Sa>	Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

## Operation

```

CheckFPEEnabled64();

bits(esize) operandA = V[a, esize];
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRTType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a, 128] else Zeros(128);

operand1 = FPNeg(operand1);
Elem[result, 0, esize] = FPMulAdd(operandA, operand1, operand2, fpcr);

V[d, 128] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMUL (by element)

Floating-point Multiply (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar, half-precision](#), [Scalar, single-precision and double-precision](#), [Vector, half-precision](#) and [Vector, single-precision and double-precision](#)

### Scalar, half-precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	0	L	M	Rm			1	0	0	1	H	0	Rn			Rd							
U																															

**FMUL** <Hd>, <Hn>, <Vm>.H[<index>]

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

### Scalar, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	1	sz	L	M	Rm			1	0	0	1	H	0	Rn			Rd							
U																															

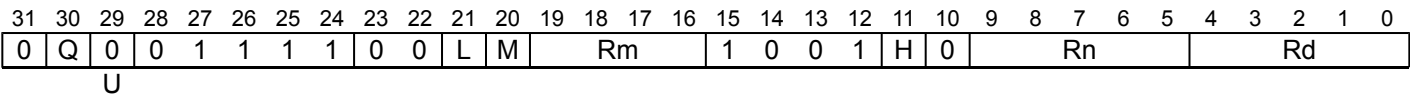
**FMUL** <V><d>, <V><n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

Vector, half-precision  
(FEAT\_FP16)



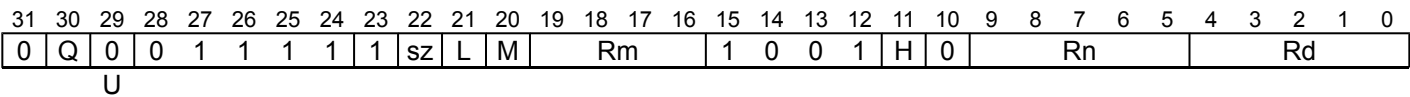
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Vector, single-precision and double-precision



FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D
- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.



<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in "sz":

sz	<Ts>
0	S
1	D

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in "sz:L:H":

sz	L	<index>
0	x	H:L
1	0	H
1	1	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxsizesize) operand2 = V[m, idxsizesize];
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if mulx_op then
        Elem[result, e, esize] = FPMulX(element1, element2, fpcr);
    else
        Elem[result, e, esize] = FPMul(element1, element2, fpcr);
V[d, 128] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

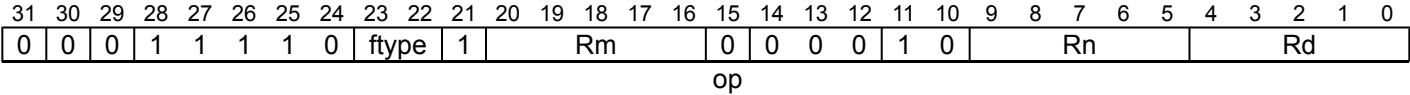
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMUL (scalar)

Floating-point Multiply (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

```
FMUL <Hd>, <Hn>, <Hm>
```

Single-precision (ftype == 00)

```
FMUL <Sd>, <Sn>, <Sm>
```

Double-precision (ftype == 01)

```
FMUL <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPEEnabled64();  
bits(esize) operand1 = V[n, esize];  
bits(esize) operand2 = V[m, esize];  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[n, 128] else Zeros(128);  
  
bits(esize) product = FPMul(operand1, operand2, fpcr);  
Elem[result, 0, esize] = product;  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMUL (vector)

Floating-point Multiply (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR EL1*, *CPTR EL2*, and *CPTR EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Half-precision* and *Single-precision and double-precision*

Half-precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	0	Rm				0	0	0	1	1	1	Rn				Rd						

FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	Rm				1	1	0	1	1	1	Rn				Rd						

FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);

V[d, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMULX

Floating-point Multiply extended. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	1	0	Rm				0	0	0	1	1	1	Rn				Rd						

**FMULX** <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	Rm				1	1	0	1	1	1	Rn				Rd						

**FMULX** <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

### Vector half precision

(FEAT\_FP16)

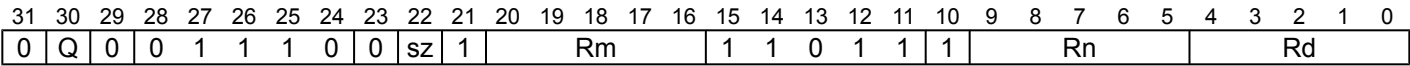
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	0	Rm				0	0	0	1	1	1	Rn				Rd						

**FMULX** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision



**FMULX** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
if elements == 1 then
    CheckFPEEnabled64();
else
    CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMulX(element1, element2, fpcr);
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## FMULX (by element)

Floating-point Multiply extended (by element). This instruction multiplies the floating-point values in the vector elements in the first source SIMD&FP register by the specified floating-point value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#) or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar, half-precision](#) , [Scalar, single-precision and double-precision](#) , [Vector, half-precision](#) and [Vector, single-precision and double-precision](#)

### Scalar, half-precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	0	L	M	Rm				1	0	0	1	H	0	Rn				Rd					
U																															

**FMULX** <Hd>, <Hn>, <Vm>.H[<index>]

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

### Scalar, single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	sz	L	M	Rm				1	0	0	1	H	0	Rn				Rd					
U																															

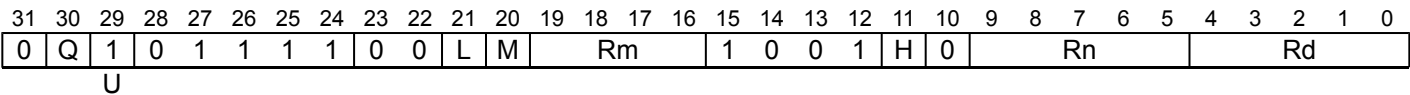
**FMULX** <V><d>, <V><n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

Vector, half-precision  
(FEAT\_FP16)



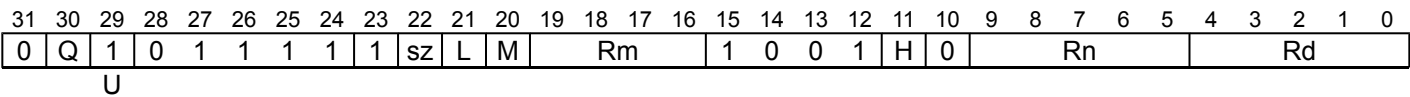
FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Vector, single-precision and double-precision



FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D
- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in "sz":

sz	<Ts>
0	S
1	D

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in "sz:L:H":

sz	L	<index>
0	x	H:L
1	0	H
1	1	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxsizesize) operand2 = V[m, idxsizesize];
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if mulx_op then
        Elem[result, e, esize] = FPMulX(element1, element2, fpcr);
    else
        Elem[result, e, esize] = FPMul(element1, element2, fpcr);
V[d, 128] = result;

```

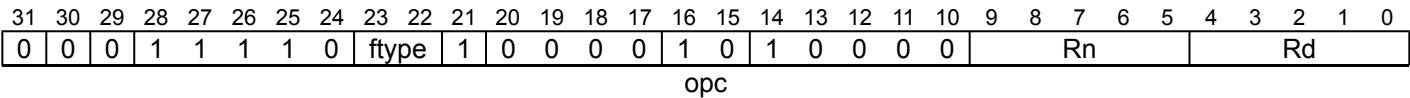
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FNEG (scalar)

Floating-point Negate (scalar). This instruction negates the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

FNEG <Hd>, <Hn>

Single-precision (ftype == 00)

FNEG <Sd>, <Sn>

Double-precision (ftype == 01)

```
FNEG <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else 0<127:0>;

bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPNeg(operand);
V[d, 128] = result;
```



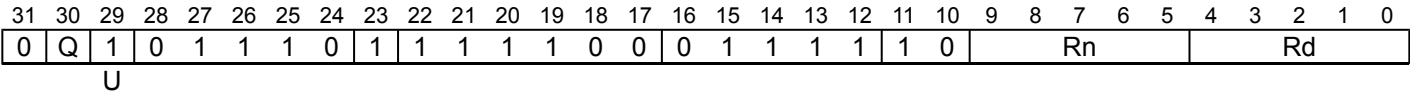
FNEG (vector)

Floating-point Negate (vector). This instruction negates the value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(FEAT\_FP16)



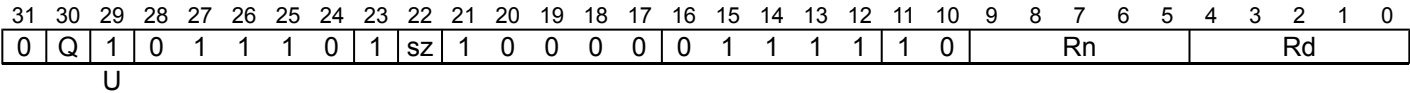
FNEG <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Single-precision and double-precision



FNEG <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;

V[d, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

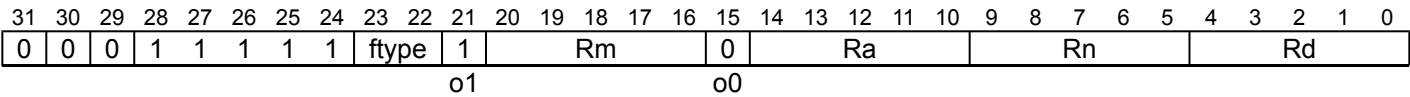
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FNMADD

Floating-point Negated fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

```
FNMADD <Hd>, <Hn>, <Hm>, <Ha>
```

Single-precision (ftype == 00)

```
FNMADD <Sd>, <Sn>, <Sm>, <Sa>
```

Double-precision (ftype == 01)

```
FNMADD <Dd>, <Dn>, <Dm>, <Da>
```

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- <Da> Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- <Ha> Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.



- <Sn> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- <Sa> Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

## Operation

```

CheckFPEEnabled64();

bits(esize) operanda = V[a, esize];
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRTType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a, 128] else Zeros(128);

operanda = FPNeg(operanda);
operand1 = FPNeg(operand1);
Elem[result, 0, esize] = FPMulAdd(operanda, operand1, operand2, fpcr);

V[d, 128] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

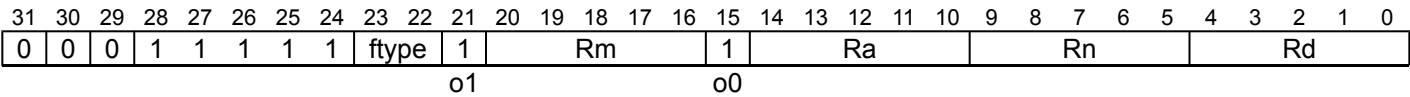
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FNMSUB

Floating-point Negated fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

```
FNMSUB <Hd>, <Hn>, <Hm>, <Ha>
```

Single-precision (ftype == 00)

```
FNMSUB <Sd>, <Sn>, <Sm>, <Sa>
```

Double-precision (ftype == 01)

```
FNMSUB <Dd>, <Dn>, <Dm>, <Da>
```

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- <Da> Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Ha>	Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Sa>	Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

## Operation

```

CheckFPEEnabled64();

bits(esize) operands = V[a, esize];
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a, 128] else Zeros(128);

operands = FPNeg(operands);
Elem[result, 0, esize] = FPMulAdd(operands, operand1, operand2, fpcr);

V[d, 128] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

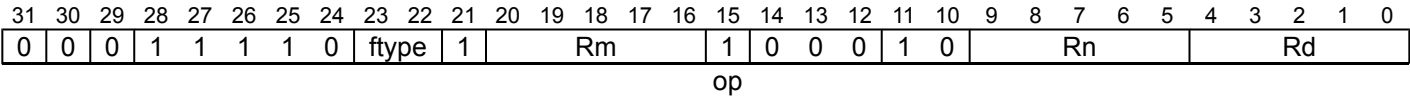
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FNMUL (scalar)

Floating-point Multiply-Negate (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the negation of the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

```
FNMUL <Hd>, <Hn>, <Hm>
```

Single-precision (ftype == 00)

```
FNMUL <Sd>, <Sn>, <Sm>
```

Double-precision (ftype == 01)

```
FNMUL <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPEnabled64();  
bits(esize) operand1 = V[n, esize];  
bits(esize) operand2 = V[m, esize];  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[n, 128] else Zeros(128);  
  
bits(esize) product = FPMul(operand1, operand2, fpcr);  
product = FPNeg(product);  
Elem[result, 0, esize] = product;  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# FRECPE

Floating-point Reciprocal Estimate. This instruction finds an approximate reciprocal estimate for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: *Scalar half precision* , *Scalar single-precision and double-precision* , *Vector half precision* and *Vector single-precision and double-precision*

## Scalar half precision (FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn				Rd					

FRECPE <Hd> , <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

## Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn				Rd					

FRECPE <V><d> , <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

## Vector half precision (FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn				Rd					

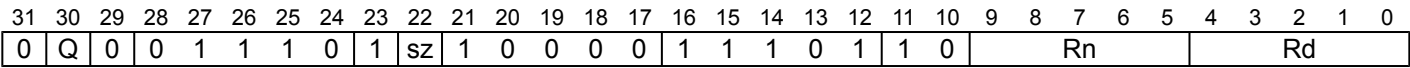
FRECPE <Vd>.<T> , <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision



FRECPE <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
if elements == 1 then
    CheckFPEEnabled64();
else
    CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRecipEstimate(element, FPCR[]);

V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## FRECPS

Floating-point Reciprocal Step. This instruction multiplies the corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 2.0, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	1	0	Rm				0	0	1	1	1	1	Rn				Rd						

**FRECPS** <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	Rm				1	1	1	1	1	1	1	Rn				Rd					

**FRECPS** <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

### Vector half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	0	Rm				0	0	1	1	1	1	1	Rn				Rd					

**FRECPS** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	sz	1	Rm					1	1	1	1	1	1	Rn					Rd				

FRECPS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
if elements == 1 then
    CheckFPEEnabled64();
else
    CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRecipStepFused(element1, element2);

V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRECPX

Floating-point Reciprocal exponent (scalar). This instruction finds an approximate reciprocal exponent for the source SIMD&FP register and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Half-precision* and *Single-precision and double-precision*

Half-precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1	1	1	1	0	Rn				Rd					

FRECPX <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	1	1	1	0	Rn				Rd					

FRECPX <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D
- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
bits(esize) operand = V[n, esize];  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
  
Elem[result, 0, esize] = FPRecpX(operand, fpcr);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINT32X (scalar)

Floating-point Round to 32-bit Integer, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 32-bit integer size using the rounding mode that is determined by the *FPCR*, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point  
(FEAT\_FRINTTS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	0	x	1	0	1	0	0	0	1	1	0	0	0	0	Rn				Rd					
ftype										op																					

Single-precision (ftype == 00)

```
FRINT32X <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINT32X <Dd>, <Dn>
```

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '1x' UNDEFINED;

FPRounding rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
bits(esize) operand = V[n, esize];  
  
Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, rounding, 32);  
  
V[d, 128] = result;
```

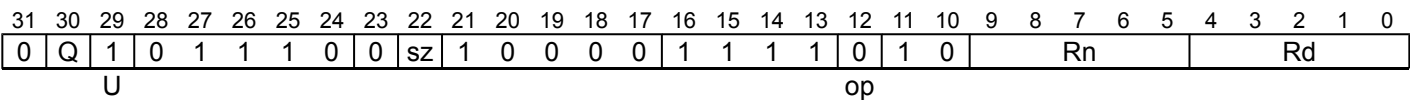
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINT32X (vector)

Floating-point Round to 32-bit Integer, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the rounding mode that is determined by the *FPCR*, and writes the result to the SIMD&FP destination register. A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised. A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*. Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision  
(FEAT\_FRINTTS)



```
FRINT32X <Vd>.<T>, <Vn>.<T>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);
V[d, datasize] = result;
```





FRINT32Z (scalar)

Floating-point Round to 32-bit Integer toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 32-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the {corresponding} input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point  
(FEAT\_FRINTTS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	0	x	1	0	1	0	0	0	0	1	0	0	0	0	Rn				Rd					
ftype										op																					

Single-precision (ftype == 00)

```
FRINT32Z <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINT32Z <Dd>, <Dn>
```

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '1x' UNDEFINED;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, FPRounding_ZERO, 32);

V[d, 128] = result;
```



FRINT32Z (vector)

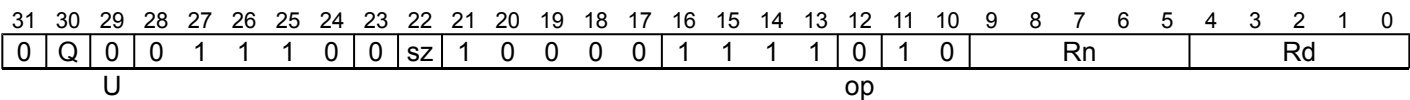
Floating-point Round to 32-bit Integer toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision  
(FEAT\_FRINTTS)



```
FRINT32Z <Vd>.<T>, <Vn>.<T>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);
V[d, datasize] = result;
```



FRINT64X (scalar)

Floating-point Round to 64-bit Integer, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the rounding mode that is determined by the *FPCR*, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point  
(FEAT\_FRINTTS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	0	x	1	0	1	0	0	1	1	1	0	0	0	0	Rn				Rd					
ftype										op																					

Single-precision (ftype == 00)

FRINT64X <Sd>, <Sn>

Double-precision (ftype == 01)

FRINT64X <Dd>, <Dn>

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '1x' UNDEFINED;

FPRounding rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
bits(esize) operand = V[n, esize];  
  
Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, rounding, 64);  
  
V[d, 128] = result;
```

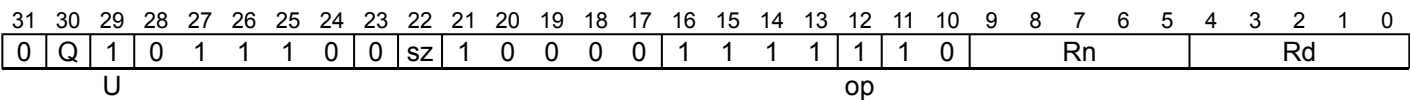
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINT64X (vector)

Floating-point Round to 64-bit Integer, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the rounding mode that is determined by the *FPCR*, and writes the result to the SIMD&FP destination register. A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised. A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*. Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision  
(FEAT\_FRINTTS)



```
FRINT64X <Vd>.<T>, <Vn>.<T>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);
V[d, datasize] = result;
```





FRINT64Z (scalar)

Floating-point Round to 64-bit Integer toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the {corresponding} input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point  
(FEAT\_FRINTTS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	0	x	1	0	1	0	0	1	0	1	0	0	0	0	Rn				Rd					
ftype										op																					

Single-precision (ftype == 00)

```
FRINT64Z <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINT64Z <Dd>, <Dn>
```

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '1x' UNDEFINED;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, FPRounding_ZERO, 64);

V[d, 128] = result;
```



FRINT64Z (vector)

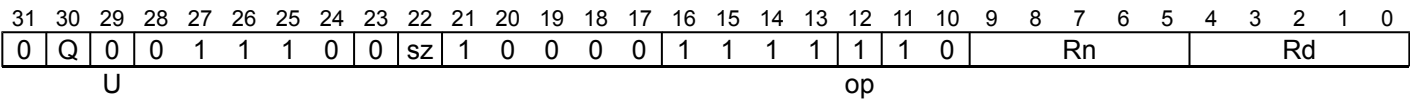
Floating-point Round to 64-bit Integer toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision  
(FEAT\_FRINTTS)



```
FRINT64Z <Vd>.<T>, <Vn>.<T>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);
V[d, datasize] = result;
```



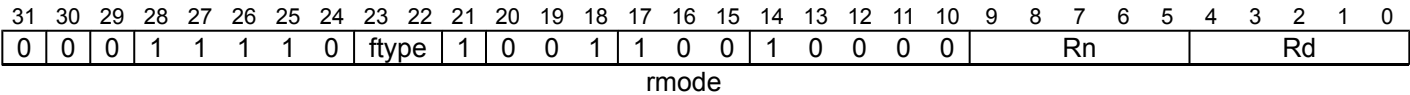
FRINTA (scalar)

Floating-point Round to Integral, to nearest with ties to Away (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

```
FRINTA <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FRINTA <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINTA <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRTType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
bits(esize) operand = V[n, esize];  
  
Elem[result, 0, esize] = FPRoundInt(operand, fpcr, FPRounding\_TIEAWAY, FALSE);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FRINTA (vector)

Floating-point Round to Integral, to nearest with ties to Away (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

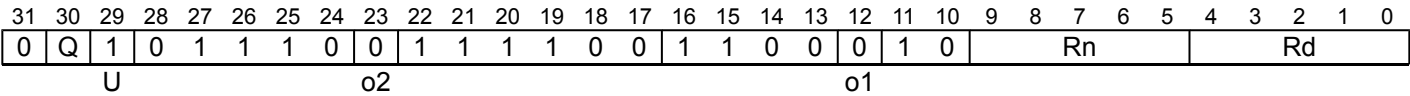
A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

### Half-precision

(FEAT\_FP16)



FRINTA <Vd>.<T>, <Vn>.<T>

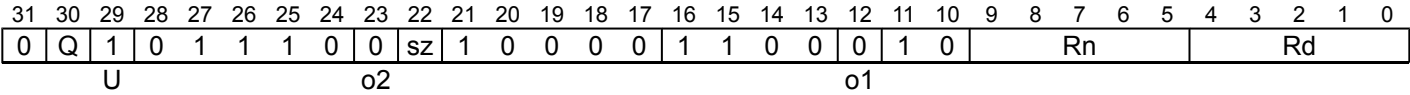
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);
```

### Single-precision and double-precision





FRINTA <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

V[d, datasize] = result;
```

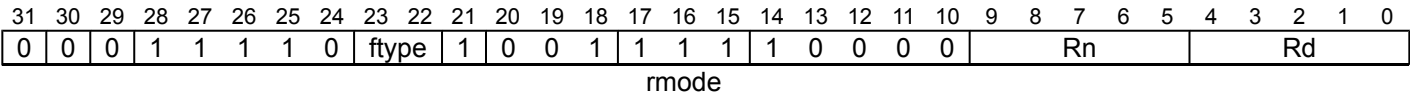
FRINTI (scalar)

Floating-point Round to Integral, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the *FPCR*, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

```
FRINTI <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FRINTI <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINTI <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

FPRounding rounding;
rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
bits(esize) operand = V[n, esize];  
  
Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTI (vector)

Floating-point Round to Integral, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the [FPCR](#), and writes the result to the SIMD&FP destination register.

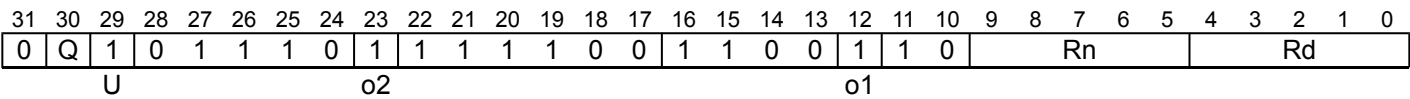
A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(FEAT\_FP16)



FRINTI <Vd>.<T>, <Vn>.<T>

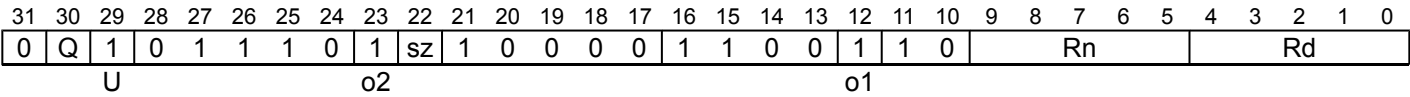
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);
```

Single-precision and double-precision



FRINTI <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

V[d, datasize] = result;
```

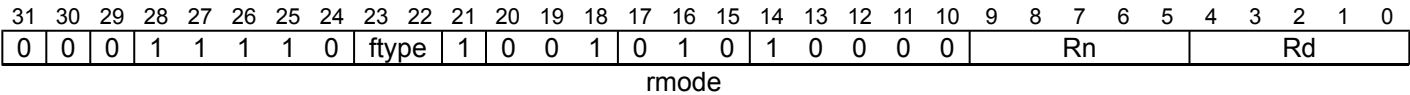
FRINTM (scalar)

Floating-point Round to Integral, toward Minus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

```
FRINTM <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FRINTM <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINTM <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

FPRounding rounding;
rounding = FPDecodeRounding('10');
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
bits(esize) operand = V[n, esize];  
  
Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTM (vector)

Floating-point Round to Integral, toward Minus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

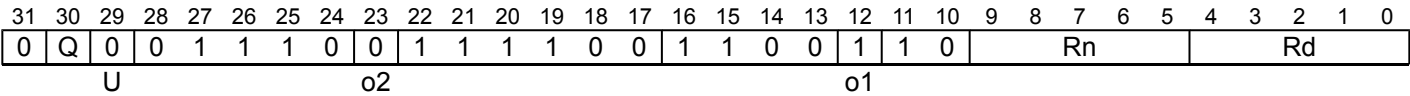
A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(FEAT\_FP16)



FRINTM <Vd>.<T>, <Vn>.<T>

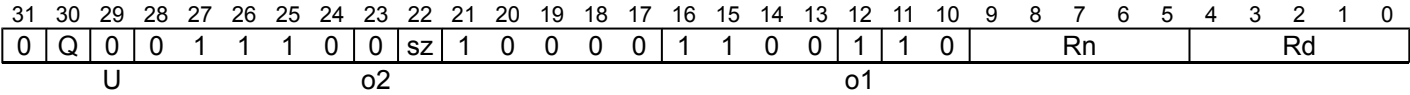
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);
```

Single-precision and double-precision





FRINTM <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

V[d, datasize] = result;
```

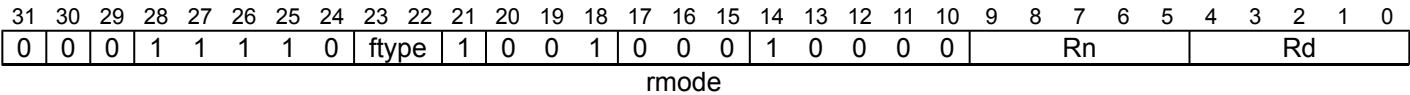
FRINTN (scalar)

Floating-point Round to Integral, to nearest with ties to even (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

```
FRINTN <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FRINTN <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINTN <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

FPRounding rounding;
rounding = FPDecodeRounding('00');
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
bits(esize) operand = V[n, esize];  
  
Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTN (vector)

Floating-point Round to Integral, to nearest with ties to even (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

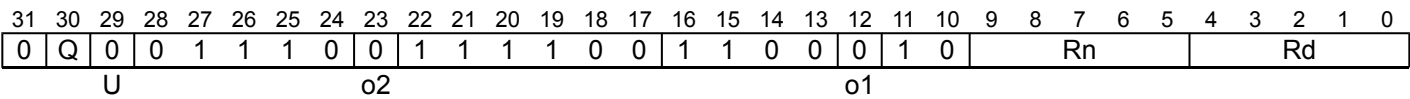
A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(FEAT\_FP16)



FRINTN <Vd>.<T>, <Vn>.<T>

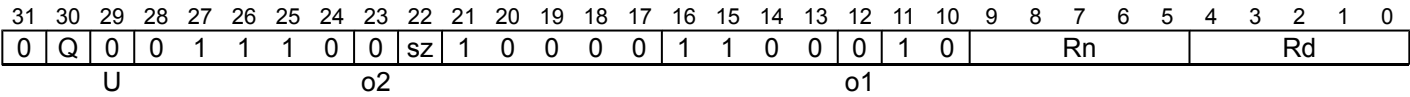
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);
```

Single-precision and double-precision



FRINTN <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

V[d, datasize] = result;
```

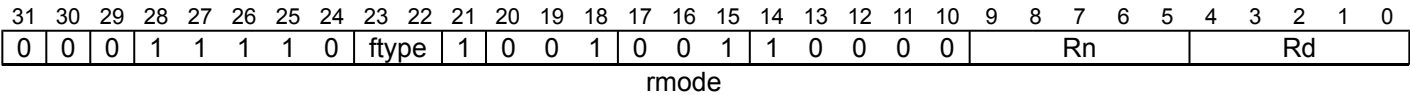
FRINTP (scalar)

Floating-point Round to Integral, toward Plus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

```
FRINTP <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FRINTP <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINTP <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

FPRounding rounding;
rounding = FPDecodeRounding('01');
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
bits(esize) operand = V[n, esize];  
  
Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTP (vector)

Floating-point Round to Integral, toward Plus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

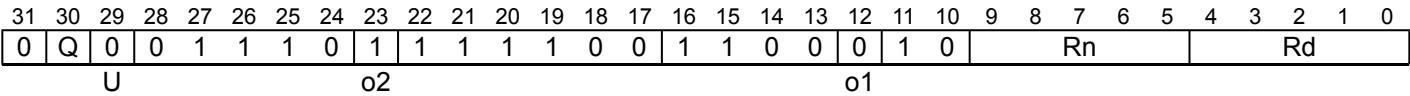
A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(FEAT\_FP16)



FRINTP <Vd>.<T>, <Vn>.<T>

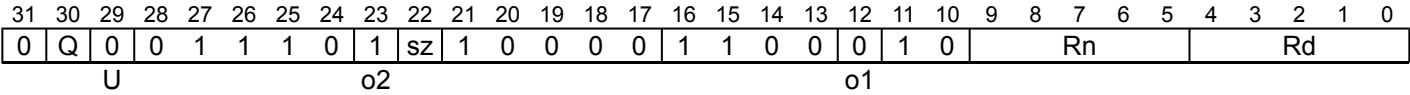
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);
```

Single-precision and double-precision





FRINTP <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

V[d, datasize] = result;
```

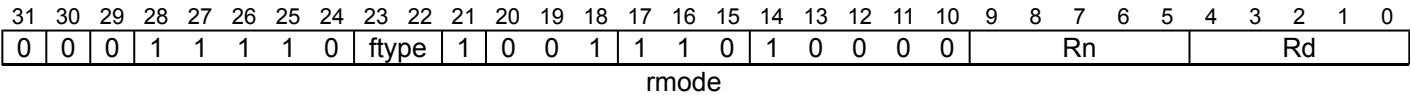
FRINTX (scalar)

Floating-point Round to Integral exact, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the *FPCR*, and writes the result to the SIMD&FP destination register.

When the result value is not numerically equal to the input value, an Inexact exception is raised. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

```
FRINTX <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FRINTX <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINTX <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;

FPRounding rounding;
rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRTType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
bits(esize) operand = V[n, esize];  
  
Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, TRUE);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTX (vector)

Floating-point Round to Integral exact, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the [FPCR](#), and writes the result to the SIMD&FP destination register.

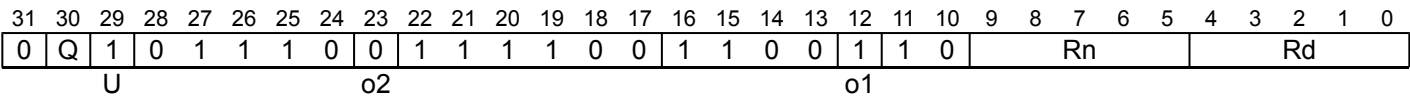
When a result value is not numerically equal to the corresponding input value, an Inexact exception is raised. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(FEAT\_FP16)



FRINTX <Vd>.<T>, <Vn>.<T>

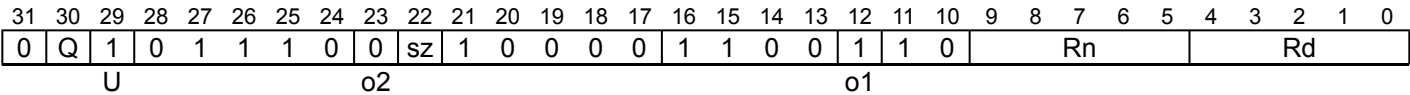
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);
```

Single-precision and double-precision



FRINTX <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

V[d, datasize] = result;
```

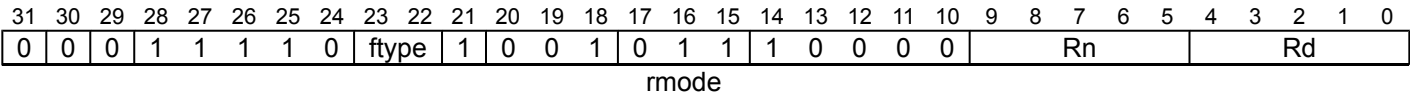
FRINTZ (scalar)

Floating-point Round to Integral, toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

```
FRINTZ <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FRINTZ <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINTZ <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

FPRounding rounding;
rounding = FPDecodeRounding('11');
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else Zeros(128);  
bits(esize) operand = V[n, esize];  
  
Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTZ (vector)

Floating-point Round to Integral, toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

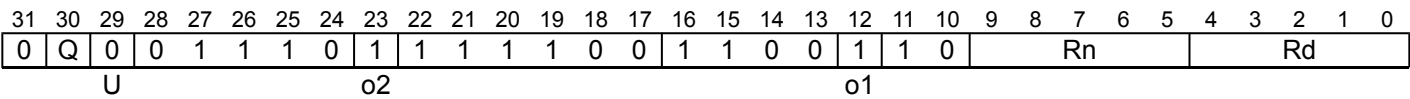
A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Half-precision* and *Single-precision and double-precision*

Half-precision  
(FEAT\_FP16)



FRINTZ <Vd>.<T>, <Vn>.<T>

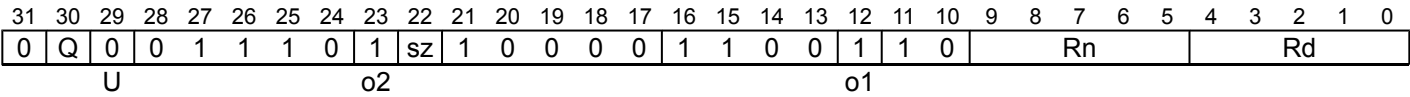
```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);
```

Single-precision and double-precision





FRINTZ <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

V[d, datasize] = result;
```

FRSQRTE

Floating-point Reciprocal Square Root Estimate. This instruction calculates an approximate square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

Scalar half precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn				Rd					

FRSQRTE <Hd> , <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn				Rd					

FRSQRTE <V><d> , <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

Vector half precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn				Rd					

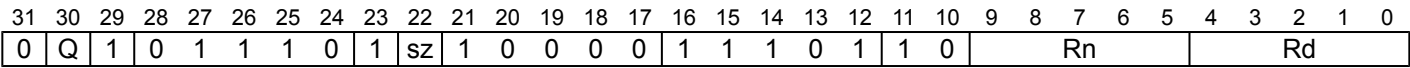
FRSQRTE <Vd>.<T> , <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision



FRSQRTE <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
if elements == 1 then
    CheckFPEEnabled64();
else
    CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRSqrtEstimate(element, fpcr);

V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FRSQRTS

Floating-point Reciprocal Square Root Step. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 3.0, divides these results by 2.0, places the results into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

### Scalar half precision

(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	1	1	1	1	0	1	1	0	Rm				0	0	1	1	1	1	Rn				Rd							

**FRSQRTS** <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

### Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	0	1	1	1	1	0	1	sz	1	Rm				1	1	1	1	1	1	1	Rn				Rd							

**FRSQRTS** <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

### Vector half precision

(FEAT\_FP16)

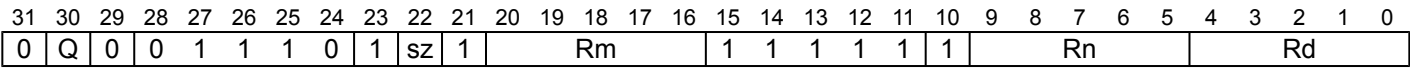
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	0	Rm				0	0	1	1	1	1	Rn				Rd						

**FRSQRTS** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision



FRSQRTS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
if elements == 1 then
    CheckFPEEnabled64();
else
    CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRSqrtStepFused(element1, element2);

V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

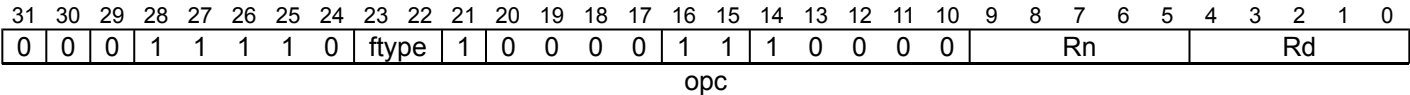
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FSQRT (scalar)

Floating-point Square Root (scalar). This instruction calculates the square root of the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

FSQRT <Hd>, <Hn>

Single-precision (ftype == 00)

FSQRT <Sd>, <Sn>

Double-precision (ftype == 01)

FSQRT <Dd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.



## Operation

```
CheckFPEnabled64();  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[d, 128] else 0<127:0>;  
  
bits(esize) operand = V[n, esize];  
  
Elem[result, 0, esize] = FPSqrt(operand, fpcr);  
  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FSQRT (vector)

Floating-point Square Root (vector). This instruction calculates the square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Half-precision* and *Single-precision and double-precision*

Half-precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	1	1	1	1	0	Rn				Rd					

FSQRT <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	1	1	1	0	Rn				Rd					

FSQRT <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPSqrt(element, FPCR[]);
V[d, datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

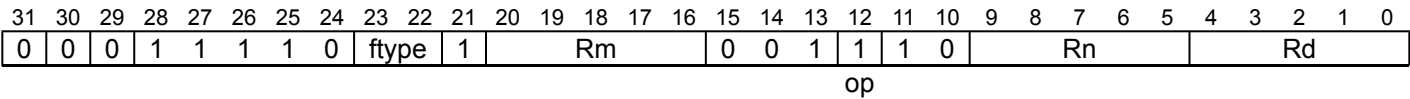
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FSUB (scalar)

Floating-point Subtract (scalar). This instruction subtracts the floating-point value of the second source SIMD&FP register from the floating-point value of the first source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)  
(FEAT\_FP16)

```
FSUB <Hd>, <Hn>, <Hm>
```

Single-precision (ftype == 00)

```
FSUB <Sd>, <Sn>, <Sm>
```

Double-precision (ftype == 01)

```
FSUB <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

Assembler Symbols

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPEEnabled64();  
bits(esize) operand1 = V[n, esize];  
bits(esize) operand2 = V[m, esize];  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
bits(128) result = if merge then V[n, 128] else Zeros(128);  
  
Elem[result, 0, esize] = FPSub(operand1, operand2, fpcr);  
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FSUB (vector)

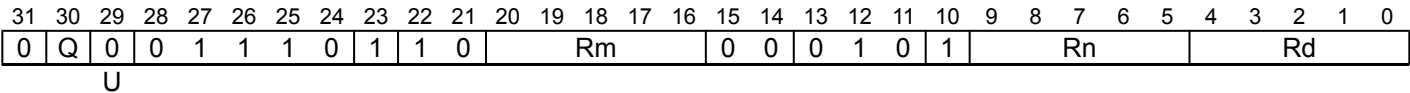
Floating-point Subtract (vector). This instruction subtracts the elements in the vector in the second source SIMD&FP register, from the corresponding elements in the vector in the first source SIMD&FP register, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR EL1*, *CPTR EL2*, and *CPTR EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision and double-precision](#)

Half-precision  
(FEAT\_FP16)

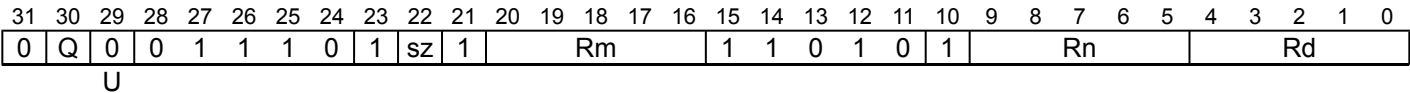


FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

Single-precision and double-precision



FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
bits(esize) diff;
FPCRTType fpcr = FPCR[];
bits(datasize) result;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, fpcr);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

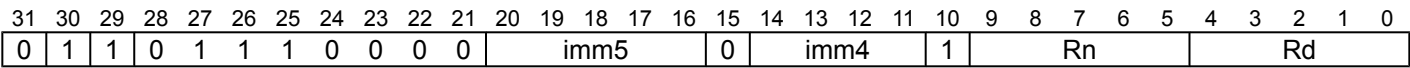
INS (element)

Insert vector element from another vector element. This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (element).



INS <Vd>.<Ts>[<index1>] , <Vn>.<Ts>[<index2>]

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer dst_index = UInt(imm5<4:size+1>);
integer src_index = UInt(imm4<3:size>);
integer idxdsize = if imm4<3> == '1' then 128 else 64;
// imm4<size-1:0> is IGNORED

integer esize = 8 << size;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in "imm5":

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index1> Is the destination element index encoded in "imm5":

imm5	<index1>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index2> Is the source element index encoded in "imm5:imm4":

imm5	<index2>
x0000	RESERVED
xxxx1	imm4<3:0>
xxx10	imm4<3:1>
xx100	imm4<3:2>
x1000	imm4<3>

Unspecified bits in "imm4" are ignored but should be set to zero by an assembler.



## Operation

```
CheckFPAdvSIMDEnabled64();  
bits(idxsize) operand = V[n, idxsize];  
bits(128) result;  
  
result = V[d, 128];  
Elem[result, dst_index, esize] = Elem[operand, src_index, esize];  
V[d, 128] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

INS (general)

Insert vector element from general-purpose register. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias [MOV \(from general\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	0	0	imm5					0	0	0	1	1	1	Rn					Rd				

INS <Vd>.<Ts>[<index>], <R><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);

if size > 3 then UNDEFINED;
integer index = UInt(imm5<4:size+1>);

integer esize = 8 << size;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in "imm5":

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index> Is the element index encoded in "imm5":

imm5	<index>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

<R> Is the width specifier for the general-purpose source register, encoded in "imm5":

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();  
bits(esize) element = X[n, esize];  
bits(128) result;  
  
result = V[d, 128];  
Elem[result, index, esize] = element;  
V[d, 128] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1 (multiple structures)

Load multiple single-element structures to one, two, three, or four registers. This instruction loads multiple single-element structures from memory and writes the result to one, two, three, or four SIMD&FP registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	x	x	1	x	size	Rn				Rt						
L										opcode																					

### One register (opcode == 0111)

```
LD1 { <Vt>.<T> }, [<Xn|SP>]
```

### Two registers (opcode == 1010)

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

### Three registers (opcode == 0110)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

### Four registers (opcode == 0010)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	1	0	Rm				x	x	1	x	size	Rn				Rt							
L										opcode																					

One register, immediate offset (Rm == 11111 && opcode == 0111)

```
LD1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

One register, register offset (Rm != 11111 && opcode == 0111)

```
LD1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

Two registers, immediate offset (Rm == 11111 && opcode == 1010)

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Two registers, register offset (Rm != 11111 && opcode == 1010)

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Three registers, immediate offset (Rm == 11111 && opcode == 0110)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Three registers, register offset (Rm != 11111 && opcode == 0110)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Four registers, immediate offset (Rm == 11111 && opcode == 0010)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Four registers, register offset (Rm != 11111 && opcode == 0010)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#8
1	#16

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#16
1	#32

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#24
1	#48

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp\_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, accdesc];
                V[tt, datasize] = rval;
            else // memop == MemOp\_STORE
                Mem[address+offs, ebytes, accdesc] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

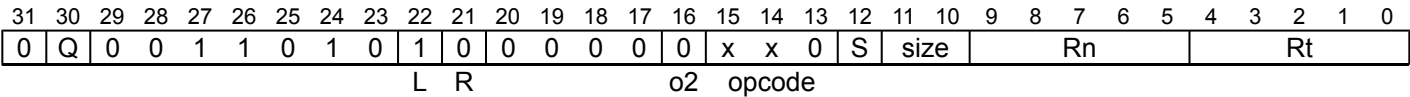
LD1 (single structure)

Load one single-element structure to one lane of one register. This instruction loads a single-element structure from memory and writes the result to the specified lane of the SIMD&FP register without affecting the other bits of the register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset



8-bit (opcode == 000)

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 010 && size == x0)

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 100 && size == 00)

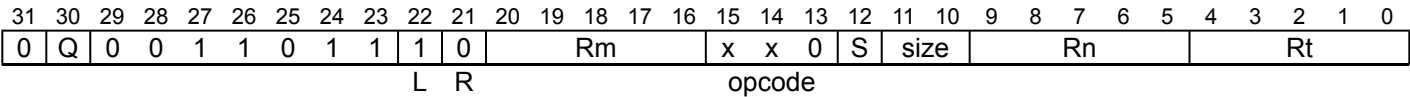
```
LD1 { <Vt>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 100 && S == 0 && size == 01)

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Post-index





**8-bit, immediate offset (Rm == 11111 && opcode == 000)**

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>], #1
```

**8-bit, register offset (Rm != 11111 && opcode == 000)**

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>
```

**16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)**

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>], #2
```

**16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)**

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>
```

**32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)**

```
LD1 { <Vt>.S }[<index>], [<Xn|SP>], #4
```

**32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)**

```
LD1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>
```

**64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)**

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>], #8
```

**64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)**

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

**Assembler Symbols**

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);    // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp\_LOAD else MemOp\_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp\_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, accdesc];
            V[t, 128] = rval;
        else // memop == MemOp\_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, accdesc] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

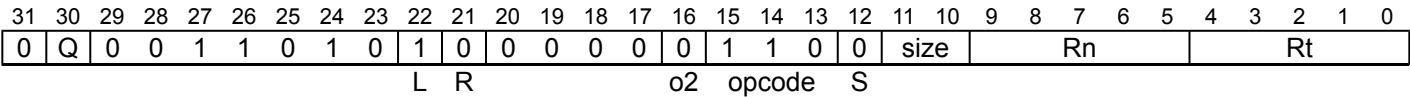
LD1R

Load one single-element structure and Replicate to all lanes (of one register). This instruction loads a single-element structure from memory and replicates the structure to all the lanes of the SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

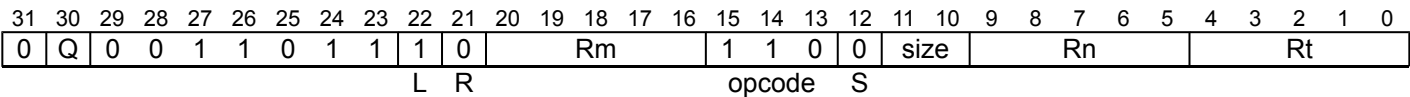
No offset



```
LD1R { <Vt>.<T> }, [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

```
LD1R { <Vt>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD1R { <Vt>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Assembler Symbols

- <Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- <T>

Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in "size":

size	<imm>
00	#1
01	#2
10	#4
11	#8

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);  // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp\_LOAD else MemOp\_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp\_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, accdesc];
            V[t, 128] = rval;
        else // memop == MemOp\_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, accdesc] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD2 (multiple structures)

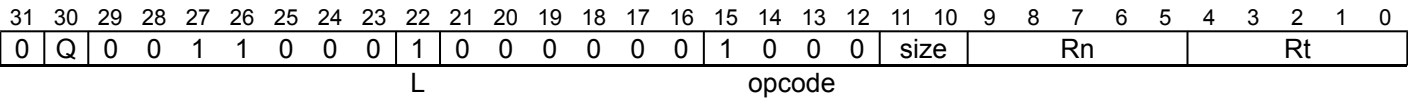
Load multiple 2-element structures to two registers. This instruction loads multiple 2-element structures from memory and writes the result to the two SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see LD3 (multiple structures).

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

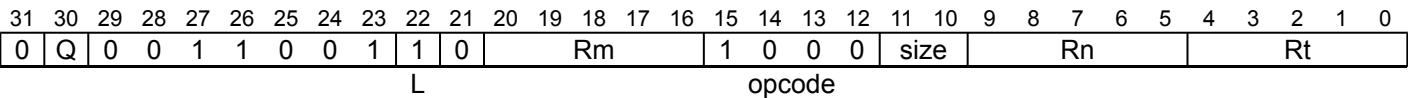
No offset



```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the post-index immediate offset, encoded in "Q":
 

Q	<imm>
0	#16
1	#32
- <Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;      // number of iterations
integer selem;    // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;      // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;      // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;      // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;      // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;      // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;      // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;      // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;

```



## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp\_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, accdesc];
                V[tt, datasize] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, accdesc] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD2 (single structure)

Load single 2-element structure to one lane of two registers. This instruction loads a 2-element structure from memory and writes the result to the corresponding elements of the two SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0	0	x	x	0	S	size	Rn				Rt						
L										R	o2										opcode										

8-bit (opcode == 000)

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 010 && size == x0)

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 100 && size == 00)

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 100 && S == 0 && size == 01)

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	1	Rm				x	x	0	S	size	Rn				Rt							
L										R	opcode																				

### 8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2
```

### 8-bit, register offset (Rm != 11111 && opcode == 000)

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4
```

### 16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8
```

### 32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16
```

### 64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp\_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, accdesc];
            V[t, 128] = rval;
        else // memop == MemOp\_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, accdesc] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

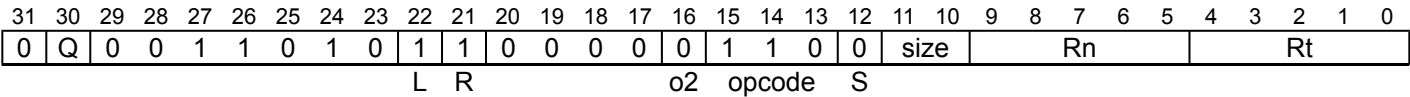
LD2R

Load single 2-element structure and Replicate to all lanes of two registers. This instruction loads a 2-element structure from memory and replicates the structure to all the lanes of the two SIMD&FP registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

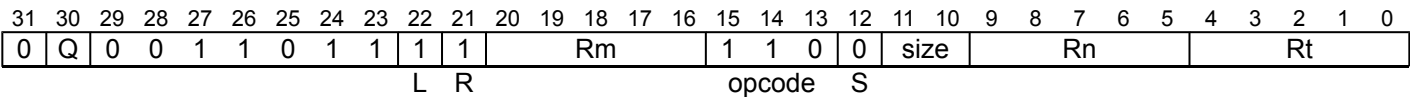
No offset



```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Assembler Symbols

- <Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- <T>

Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D
- <Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

- <Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm>** Is the post-index immediate offset, encoded in "size":
- | size | <imm> |
|------|-------|
| 00   | #2    |
| 01   | #4    |
| 10   | #8    |
| 11   | #16   |
- <Xm>** Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```

integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;

```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp\_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, accdesc];
            V[t, 128] = rval;
        else // memop == MemOp\_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, accdesc] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

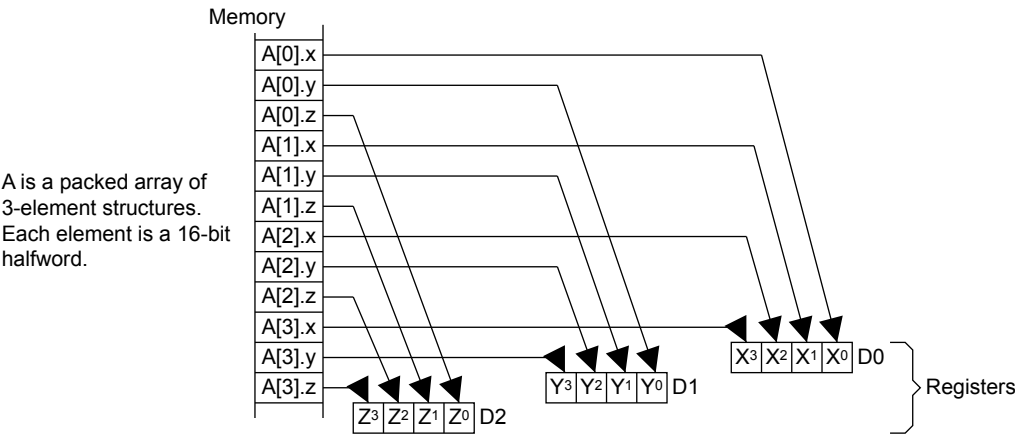
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



LD3 (multiple structures)

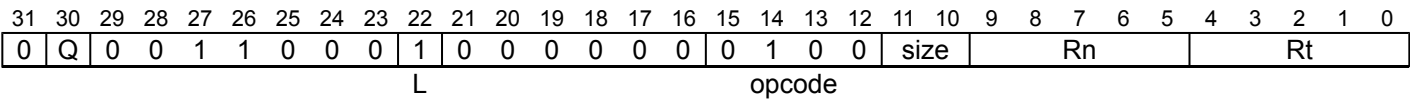
Load multiple 3-element structures to three registers. This instruction loads multiple 3-element structures from memory and writes the result to the three SIMD&FP registers, with de-interleaving. The following figure shows an example of the operation of de-interleaving of a LD3.16 (multiple 3-element structures) instruction:.



Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

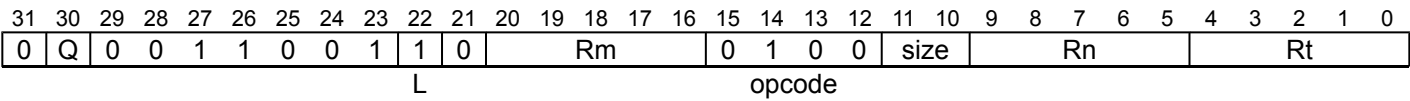
No offset



```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Assembler Symbols

- <Vt>Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- <T>Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vt2>Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- <Vt3>Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Xn|SP>Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm>Is the post-index immediate offset, encoded in “Q”:
- | Q | <imm> |
|---|-------|
| 0 | #24   |
| 1 | #48   |
- <Xm>Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp\_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, accdesc];
                V[tt, datasize] = rval;
            else // memop == MemOp\_STORE
                Mem[address+offs, ebytes, accdesc] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD3 (single structure)

Load single 3-element structure to one lane of three registers. This instruction loads a 3-element structure from memory and writes the result to the corresponding elements of the three SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	0	0	0	0	0	0	x	x	1	S	size	Rn				Rt						
L R										o2 opcode																					

8-bit (opcode == 001)

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 101 && size == 00)

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 101 && S == 0 && size == 01)

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	0	Rm				x	x	1	S	size	Rn				Rt							
L R										opcode																					

### 8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3
```

### 8-bit, register offset (Rm != 11111 && opcode == 001)

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6
```

### 16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12
```

### 32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24
```

### 64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp\_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, accdesc];
            V[t, 128] = rval;
        else // memop == MemOp\_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, accdesc] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

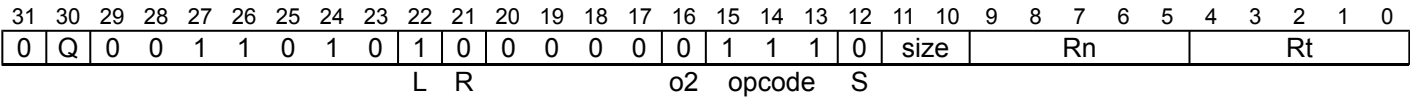
LD3R

Load single 3-element structure and Replicate to all lanes of three registers. This instruction loads a 3-element structure from memory and replicates the structure to all the lanes of the three SIMD&FP registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

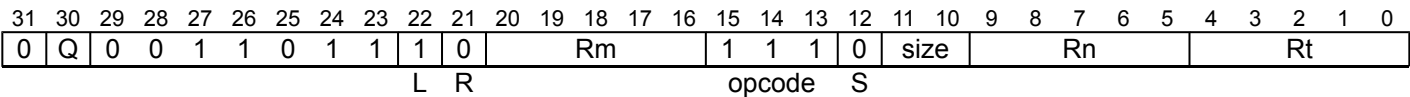
No offset



LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Assembler Symbols

- <Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D
- <Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.



- <Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the post-index immediate offset, encoded in "size":
 

size	<imm>
00	#3
01	#6
10	#12
11	#24
- <Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
      scale = 3;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp\_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, accdesc];
            V[t, 128] = rval;
        else // memop == MemOp\_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, accdesc] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD4 (multiple structures)

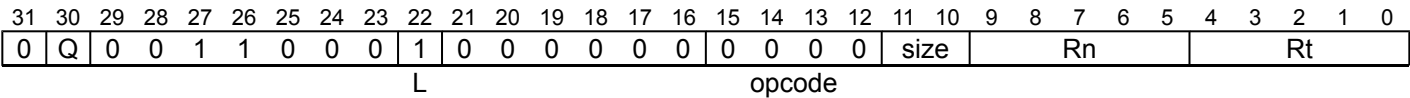
Load multiple 4-element structures to four registers. This instruction loads multiple 4-element structures from memory and writes the result to the four SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see LD3 (multiple structures).

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

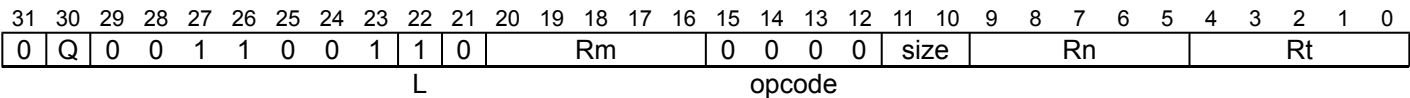
No offset



```
LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

```
LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Assembler Symbols

- <Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- <Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the post-index immediate offset, encoded in "Q":
 

Q	<imm>
0	#32
1	#64
- <Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;      // number of iterations
integer selem;    // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;      // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;      // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;      // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;      // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;      // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;      // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;      // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;

```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp\_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, accdesc];
                V[tt, datasize] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, accdesc] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD4 (single structure)

Load single 4-element structure to one lane of four registers. This instruction loads a 4-element structure from memory and writes the result to the corresponding elements of the four SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0	0	x	x	1	S	size	Rn				Rt						
L										R	o2										opcode										

8-bit (opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 101 && size == 00)

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 101 && S == 0 && size == 01)

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	1	1	Rm				x	x	1	S	size	Rn				Rt							
L										R	opcode																				

### 8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

### 8-bit, register offset (Rm != 11111 && opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

### 16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

### 32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

### 64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4>	Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```



## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp\_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, accdesc];
            V[t, 128] = rval;
        else // memop == MemOp\_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, accdesc] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

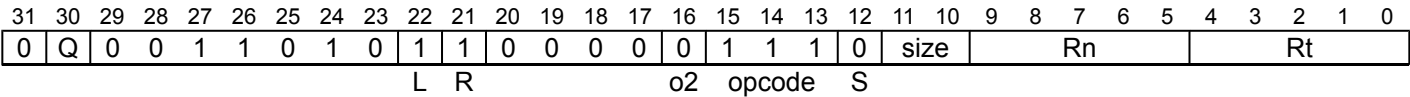
LD4R

Load single 4-element structure and Replicate to all lanes of four registers. This instruction loads a 4-element structure from memory and replicates the structure to all the lanes of the four SIMD&FP registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

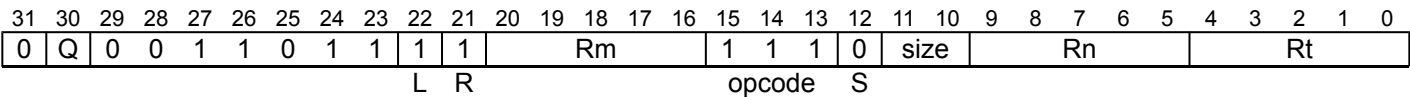
No offset



```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Assembler Symbols

- <Vt>

Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- <T>

Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D
- <Vt2>

Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

- <Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the post-index immediate offset, encoded in "size":
 

size	<imm>
00	#4
01	#8
10	#16
11	#32
- <Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```

integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;

```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp\_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, accdesc];
            V[t, 128] = rval;
        else // memop == MemOp\_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, accdesc] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDAP1 (SIMD&FP)

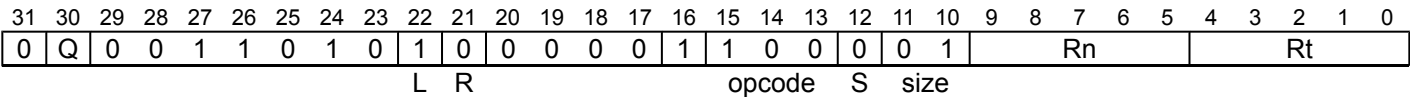
Load-Acquire RCpc one single-element structure to one lane of one register. This instruction loads a single-element structure from memory and writes the result to the specified lane of the SIMD&FP register without affecting the other bits of the register.

The instruction has memory ordering semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.  
For information about memory accesses, see *Load/Store addressing modes*.  
Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

64-bit  
(FEAT\_LRCPC3)



```
LDAP1 { <Vt>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Assembler Symbols

- <Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- <index> Is the element index, encoded in "Q".
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp\_LOAD else MemOp\_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMDAcqRel(memop, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp\_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, accdesc];
            V[t, 128] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, accdesc] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDAPUR (SIMD&FP)

Load-Acquire RCpc SIMD&FP Register (unscaled offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset. The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode. Depending on the settings in the *CPACR EL1*, *CPTR EL2*, and *CPTR EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Unscaled offset  
(FEAT\_LRCPC3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	0	1	1	1	0	1	x	1	0	imm9										1	0	Rn				Rt					
opc																															

8-bit (size == 00 && opc == 01)

```
LDAPUR <Bt>, [<Xn|SP>{, #<sim>}]
```

16-bit (size == 01 && opc == 01)

```
LDAPUR <Ht>, [<Xn|SP>{, #<sim>}]
```

32-bit (size == 10 && opc == 01)

```
LDAPUR <St>, [<Xn|SP>{, #<sim>}]
```

64-bit (size == 11 && opc == 01)

```
LDAPUR <Dt>, [<Xn|SP>{, #<sim>}]
```

128-bit (size == 00 && opc == 11)

```
LDAPUR <Qt>, [<Xn|SP>{, #<sim>}]
```

```
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.



## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tagchecked = memop != MemOp_PREFETCH && (n != 31);
```

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescASIMDAcqRel(memop, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t, datasize];
        Mem[address, datasize DIV 8, accdesc] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, accdesc];
        V[t, datasize] = data;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

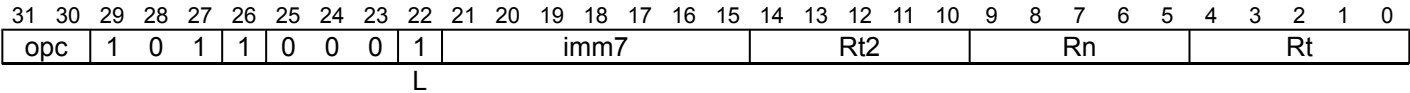
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDNP (SIMD&FP)

Load Pair of SIMD&FP registers, with Non-temporal hint. This instruction loads a pair of SIMD&FP registers from memory, issuing a hint to the memory system that the access is non-temporal. The address that is used for the load is calculated from a base register value and an optional immediate offset.

For information about non-temporal pair instructions, see *Load/Store SIMD and Floating-point Non-temporal pair*. Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



32-bit (opc == 00)

```
LDNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 01)

```
LDNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

128-bit (opc == 10)

```
LDNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

// Empty.

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDNP (SIMD&FP)*.

Assembler Symbols

- <Dt1>

Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Dt2>

Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- <Qt1>

Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Qt2>

Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- <St1>

Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- <St2>

Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm>

For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.  
For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.  
For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable\_LDPOVERLAP);
    assert c IN {Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP};
    case c of
        when Constraint\_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint\_UNDEF    UNDEFINED;
        when Constraint\_NOP      EndOfInstruction();
```

## Operation

```
CheckFPEEnabled64();
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(MemOp\_LOAD, TRUE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data1 = Mem[address, dbytes, accdesc];
data2 = Mem[address+dbytes, dbytes, accdesc];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
V[t, datasize] = data1;
V[t2, datasize] = data2;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

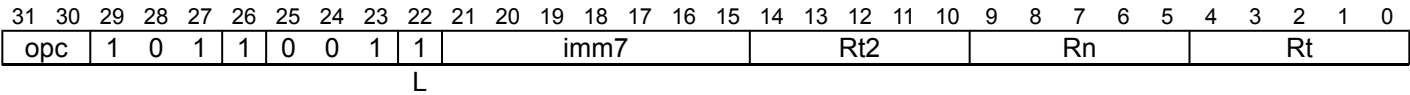
LDP (SIMD&FP)

Load Pair of SIMD&FP registers. This instruction loads a pair of SIMD&FP registers from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Signed offset](#)

Post-index



32-bit (opc == 00)

```
LDP <St1>, <St2>, [<Xn|SP>], #<imm>
```

64-bit (opc == 01)

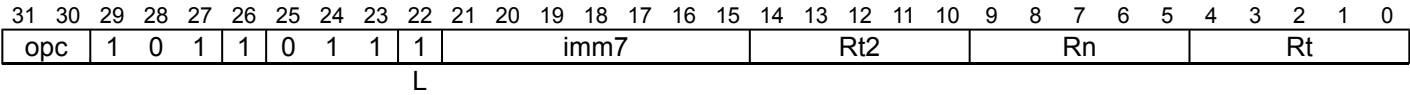
```
LDP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>
```

128-bit (opc == 10)

```
LDP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index



32-bit (opc == 00)

```
LDP <St1>, <St2>, [<Xn|SP>, #<imm>]!
```

64-bit (opc == 01)

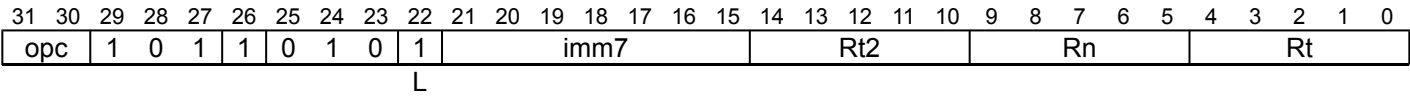
```
LDP <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]!
```

128-bit (opc == 10)

```
LDP <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset



### 32-bit (opc == 00)

```
LDP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

### 64-bit (opc == 01)

```
LDP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

### 128-bit (opc == 10)

```
LDP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;  
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [LDP \(SIMD&FP\)](#).

## Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.  For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.  For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.  For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.  For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16.  For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable\_LDPOVERLAP);
    assert c IN {Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP};
    case c of
        when Constraint\_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint\_UNDEF    UNDEFINED;
        when Constraint\_NOP      EndOfInstruction();
```

## Operation

```
CheckFPEEnabled64();
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(MemOp\_LOAD, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

data1 = Mem[address, dbytes, accdesc];
data2 = Mem[address+dbytes, dbytes, accdesc];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
V[t, datasize] = data1;
V[t2, datasize] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

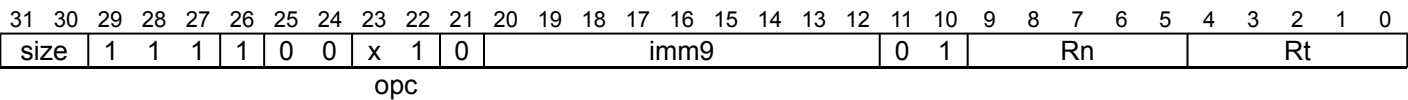
## LDR (immediate, SIMD&FP)

Load SIMD&FP Register (immediate offset). This instruction loads an element from memory, and writes the result as a scalar to the SIMD&FP register. The address that is used for the load is calculated from a base register value, a signed immediate offset, and an optional offset that is a multiple of the element size.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Unsigned offset](#)

### Post-index



#### 8-bit (size == 00 && opc == 01)

LDR <Bt>, [<Xn|SP>], #<sim>

#### 16-bit (size == 01 && opc == 01)

LDR <Ht>, [<Xn|SP>], #<sim>

#### 32-bit (size == 10 && opc == 01)

LDR <St>, [<Xn|SP>], #<sim>

#### 64-bit (size == 11 && opc == 01)

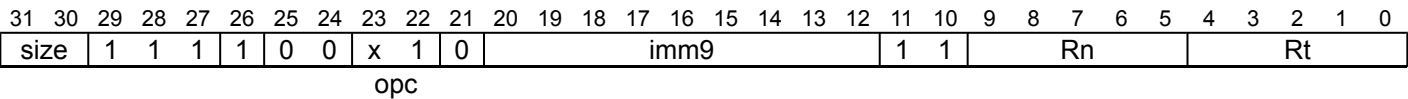
LDR <Dt>, [<Xn|SP>], #<sim>

#### 128-bit (size == 00 && opc == 11)

LDR <Qt>, [<Xn|SP>], #<sim>

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index



8-bit (size == 00 && opc == 01)

```
LDR <Bt>, [<Xn|SP>, #<simm>]!
```

16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>, #<simm>]!
```

32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>, #<simm>]!
```

64-bit (size == 11 && opc == 01)

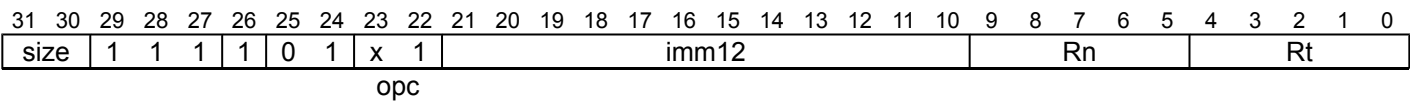
```
LDR <Dt>, [<Xn|SP>, #<simm>]!
```

128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



8-bit (size == 00 && opc == 01)

```
LDR <Bt>, [<Xn|SP>{, #<pimm>}]
```

16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>{, #<pimm>}]
```

32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>{, #<pimm>}]
```

64-bit (size == 11 && opc == 01)

```
LDR <Dt>, [<Xn|SP>{, #<pimm>}]
```

128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```



## Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.  For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.  For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.  For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.  For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as <pimm>/16.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tagchecked = memop != MemOp_PREFETCH && (wback || n != 31);
```

## Operation

```
CheckFPEEnabled64();
bits(64) address;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t, datasize];
        Mem[address, datasize DIV 8, accdesc] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, accdesc];
        V[t, datasize] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDR (literal, SIMD&FP)

Load SIMD&FP Register (PC-relative literal). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from the PC value and an immediate offset.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		0	1	1	1	0	0	imm19																		Rt					

32-bit (opc == 00)

```
LDR <St>, <label>
```

64-bit (opc == 01)

```
LDR <Dt>, <label>
```

128-bit (opc == 10)

```
LDR <Qt>, <label>
```

```
integer t = UInt(Rt);
integer size;
bits(64) offset;

case opc of
  when '00'
    size = 4;
  when '01'
    size = 8;
  when '10'
    size = 16;
  when '11'
    UNDEFINED;

offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

- <Dt> Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
- <Qt> Is the 128-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
- <St> Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
- <label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(64) address = PC[] + offset;
bits(size*8) data;

CheckFPEEnabled64();
AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, FALSE, FALSE);
data = Mem[address, size, accdesc];
V[t, size*8] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



## LDR (register, SIMD&FP)

Load SIMD&FP Register (register offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	1	1	1	1	0	0	x	1	1	Rm						option		S	1	0	Rn						Rt				
opc																															

8-bit (size == 00 && opc == 01 && option != 011)

```
LDR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

8-bit (size == 00 && opc == 01 && option == 011)

```
LDR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit (size == 11 && opc == 01)

```
LDR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

### Assembler Symbols

- <Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> For the 8-bit variant: is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	SCTX

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SCTX

<amount> For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#4

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tagchecked = memop != MemOp_PREFETCH;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
CheckFPEnabled64();
bits(64) address;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp\_STORE
        data = V[t, datasize];
        Mem[address, datasize DIV 8, accdesc] = data;

    when MemOp\_LOAD
        data = Mem[address, datasize DIV 8, accdesc];
        V[t, datasize] = data;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDUR (SIMD&FP)

Load SIMD&FP Register (unscaled offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	1	1	1	1	0	0	x	1	0	imm9										0	0	Rn					Rt				
opc																															

8-bit (size == 00 && opc == 01)

```
LDUR <Bt>, [<Xn|SP>{, #<sim>}]
```

16-bit (size == 01 && opc == 01)

```
LDUR <Ht>, [<Xn|SP>{, #<sim>}]
```

32-bit (size == 10 && opc == 01)

```
LDUR <St>, [<Xn|SP>{, #<sim>}]
```

64-bit (size == 11 && opc == 01)

```
LDUR <Dt>, [<Xn|SP>{, #<sim>}]
```

128-bit (size == 00 && opc == 11)

```
LDUR <Qt>, [<Xn|SP>{, #<sim>}]
```

```
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tagchecked = memop != MemOp_PREFETCH && (n != 31);
```



## Operation

```
CheckFPEEnabled64();  
bits(64) address;  
bits(datasize) data;  
  
AccessDescriptor accdesc = CreateAccDescASIMD(memop, FALSE, tagchecked);  
  
if n == 31 then  
    CheckSPAlignment();  
    address = SP[];  
else  
    address = X[n, 64];  
  
address = address + offset;  
  
case memop of  
    when MemOp\_STORE  
        data = V[t, datasize];  
        Mem[address, datasize DIV 8, accdesc] = data;  
  
    when MemOp\_LOAD  
        data = Mem[address, datasize DIV 8, accdesc];  
        V[t, datasize] = data;
```

## Operational information

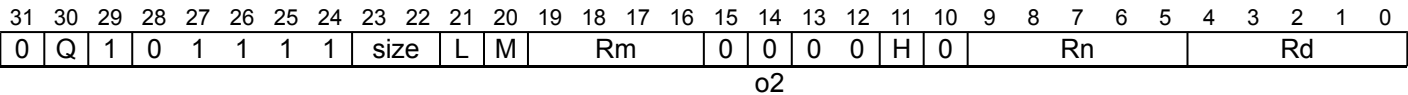
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MLA (by element)

Multiply-Add to accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxsizesize) operand2 = V[m, idxdsizesize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d, datasize] = result;
```

Operational information

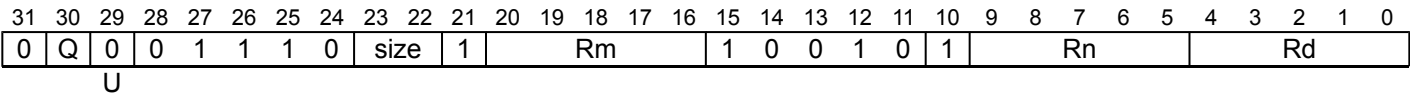
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

MLA (vector)

Multiply-Add to accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1)*UInt(element2))<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

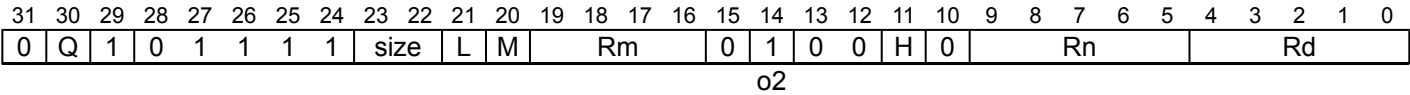
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MLS (by element)

Multiply-Subtract from accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d, datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

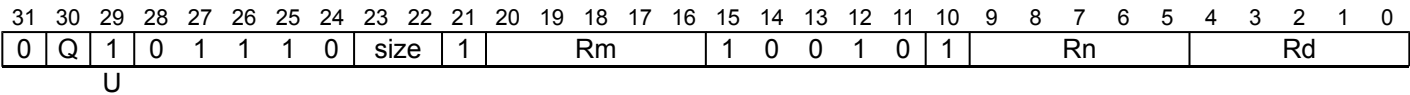
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MLS (vector)

Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1)*UInt(element2))<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;

V[d, datasize] = result;
```



## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (element)

Move vector element to another vector element. This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of [INS \(element\)](#). This means:

- The encodings in this description are named to match the encodings of [INS \(element\)](#).
- The description of [INS \(element\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	1	0	1	1	1	0	0	0	0	imm5					0	imm4					1	Rn					Rd				

MOV <Vd>.<Ts>[<index1>] , <Vn>.<Ts>[<index2>]

is equivalent to

INS <Vd>.<Ts>[<index1>] , <Vn>.<Ts>[<index2>]

and is always the preferred disassembly.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in "imm5":

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index1> Is the destination element index encoded in "imm5":

imm5	<index1>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index2> Is the source element index encoded in "imm5:imm4":

imm5	<index2>
x0000	RESERVED
xxxx1	imm4<3:0>
xxx10	imm4<3:1>
xx100	imm4<3:2>
x1000	imm4<3>

Unspecified bits in "imm4" are ignored but should be set to zero by an assembler.

Operation

The description of [INS \(element\)](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (from general)

Move general-purpose register to a vector element. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.  
This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.  
Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of [INS \(general\)](#). This means:

- The encodings in this description are named to match the encodings of [INS \(general\)](#).
- The description of [INS \(general\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	0	0	imm5					0	0	0	1	1	1	Rn					Rd				

MOV <Vd>.<Ts>[<index>], <R><n>

is equivalent to

INS <Vd>.<Ts>[<index>], <R><n>

and is always the preferred disassembly.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in "imm5":

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index> Is the element index encoded in "imm5":

imm5	<index>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

<R> Is the width specifier for the general-purpose source register, encoded in "imm5":

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

Operation

The description of [INS \(general\)](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (scalar)

Move vector element to scalar. This instruction duplicates the specified vector element in the SIMD&FP source register into a scalar, and writes the result to the SIMD&FP destination register.  
Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of [DUP \(element\)](#). This means:

- The encodings in this description are named to match the encodings of [DUP \(element\)](#).
- The description of [DUP \(element\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0	imm5					0	0	0	0	0	1	Rn					Rd				

MOV <V><d>, <Vn>.<T>[<index>]

is equivalent to

DUP <V><d>, <Vn>.<T>[<index>]

and is always the preferred disassembly.

Assembler Symbols

<V> Is the destination width specifier, encoded in “imm5”:

imm5	<V>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is the element width specifier, encoded in “imm5”:

imm5	<T>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index> Is the element index encoded in “imm5”:

imm5	<index>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

Operation

The description of [DUP \(element\)](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (to general)

Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of UMOV. This means:

- The encodings in this description are named to match the encodings of UMOV.
- The description of UMOV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	0	x	x	x	0	0	0	0	1	1	1	1	Rn				Rd					
imm5																															

32-bit (Q == 0 && imm5 == xx100)

```
MOV <Wd>, <Vn>.S[<index>]
```

is equivalent to

```
UMOV <Wd>, <Vn>.S[<index>]
```

and is always the preferred disassembly.

64-bit (Q == 1 && imm5 == x1000)

```
MOV <Xd>, <Vn>.D[<index>]
```

is equivalent to

```
UMOV <Xd>, <Vn>.D[<index>]
```

and is always the preferred disassembly.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <index> For the 32-bit variant: is the element index encoded in "imm5<4:3>".  
For the 64-bit variant: is the element index encoded in "imm5<4>".

Operation

The description of UMOV gives the operational pseudocode for this instruction.

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.





MOV (vector)

Move vector. This instruction copies the vector in the source SIMD&FP register into the destination SIMD&FP register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of ORR (vector, register). This means:

- The encodings in this description are named to match the encodings of ORR (vector, register).
- The description of ORR (vector, register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	0	1	Rm					0	0	0	1	1	1	Rn					Rd				
size																															

MOV <Vd>.<T>, <Vn>.<T>

is equivalent to

ORR <Vd>.<T>, <Vn>.<T>, <Vn>.<T>

and is the preferred disassembly when Rm == Rn.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Operation

The description of ORR (vector, register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## MOVI

Move Immediate (vector). This instruction places an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	op	0	1	1	1	1	0	0	0	0	0	a	b	c	cmode				0	1	d	e	f	g	h	Rd				

### 8-bit (op == 0 && cmode == 1110)

```
MOVI <Vd>.<T>, #<imm8>{, LSL #0}
```

### 16-bit shifted immediate (op == 0 && cmode == 10x0)

```
MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### 32-bit shifted immediate (op == 0 && cmode == 0xx0)

```
MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### 32-bit shifting ones (op == 0 && cmode == 110x)

```
MOVI <Vd>.<T>, #<imm8>, MSL #<amount>
```

### 64-bit scalar (Q == 0 && op == 1 && cmode == 1110)

```
MOVI <Dd>, #<imm>
```

### 64-bit vector (Q == 1 && op == 1 && cmode == 1110)

```
MOVI <Vd>.2D, #<imm>
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
  when '0xx00' operation = ImmediateOp MOVI;
  when '0xx01' operation = ImmediateOp MVNI;
  when '0xx10' operation = ImmediateOp ORR;
  when '0xx11' operation = ImmediateOp BIC;
  when '10x00' operation = ImmediateOp MOVI;
  when '10x01' operation = ImmediateOp MVNI;
  when '10x10' operation = ImmediateOp ORR;
  when '10x11' operation = ImmediateOp BIC;
  when '110x0' operation = ImmediateOp MOVI;
  when '110x1' operation = ImmediateOp MVNI;
  when '1110x' operation = ImmediateOp MOVI;
  when '11110' operation = ImmediateOp MOVI;
  when '11111'
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;
    operation = ImmediateOp MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

- <Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <imm>

Is a 64-bit immediate 'aaaaaaaabbbbbbbccccccddddddeeeeeeffffffffggggggghhhhhhh', encoded in "a:b:c:d:e:f:g:h".

<T> For the 8-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

For the 16-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit shifted immediate variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the 32-bit shifted immediate variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

For the 32-bit shifting ones variant: is the shift amount encoded in “cmode<0>”:

cmode<0>	<amount>
0	8
1	16

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
  when ImmediateOp_MOVI
    result = imm;
  when ImmediateOp_MVNI
    result = NOT(imm);
  when ImmediateOp_ORR
    operand = V[rd, datasize];
    result = operand OR imm;
  when ImmediateOp_BIC
    operand = V[rd, datasize];
    result = operand AND NOT(imm);

V[rd, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MUL (by element)

Multiply (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M	Rm				1	0	0	0	H	0	Rn				Rd						

MUL <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in "size:L:H:M":

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    Elem[result, e, esize] = product;

V[d, datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

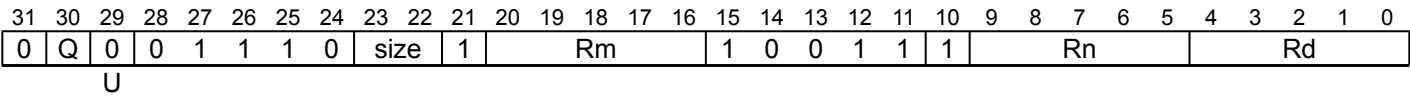
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MUL (vector)

Multiply (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



MUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean poly = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
    else
        product = (UInt(element1)*UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;

V[d, datasize] = result;
```



## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MVN

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of [NOT](#). This means:

- The encodings in this description are named to match the encodings of [NOT](#).
- The description of [NOT](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	0	1	0	0	0	0	0	0	1	0	1	1	0	Rn					Rd				

**MVN** <Vd>.<T>, <Vn>.<T>

is equivalent to

**NOT** <Vd>.<T>, <Vn>.<T>

and is always the preferred disassembly.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

The description of [NOT](#) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

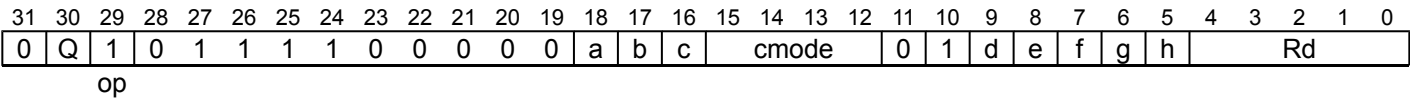
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MVNI

Move inverted Immediate (vector). This instruction places the inverse of an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



16-bit shifted immediate (cmode == 10x0)

```
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit shifted immediate (cmode == 0xx0)

```
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit shifting ones (cmode == 110x)

```
MVNI <Vd>.<T>, #<imm8>, MSL #<amount>
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
  when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx11' operation = ImmediateOp_BIC;
  when '10x01' operation = ImmediateOp_MVNI;
  when '10x11' operation = ImmediateOp_BIC;
  when '110x1' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '11111'
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;
    operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the 16-bit variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit shifted immediate variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the 32-bit shifted immediate variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

For the 32-bit shifting ones variant: is the shift amount encoded in “cmode<0>”:

cmode<0>	<amount>
0	8
1	16

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
  when ImmediateOp_MOVI
    result = imm;
  when ImmediateOp_MVNI
    result = NOT(imm);
  when ImmediateOp_ORR
    operand = V[rd, datasize];
    result = operand OR imm;
  when ImmediateOp_BIC
    operand = V[rd, datasize];
    result = operand AND NOT(imm);

V[rd, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

NEG (vector)

Negate (vector). This instruction reads each vector element from the source SIMD&FP register, negates each value, puts the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size	1	0	0	0	0	0	0	1	0	1	1	1	0	Rn				Rd					
U																															

NEG <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	0	0	1	0	1	1	1	0	Rn				Rd					
U																															

NEG <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

- <V>

Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D
- <d>

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n>

Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

NOT

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias [MVN](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	0	1	0	0	0	0	0	0	1	0	1	1	0	Rn				Rd					

NOT <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "Q":
- | Q | <T> |
|---|-----|
| 0 | 8B  |
| 1 | 16B |
- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = NOT(element);

V[d, datasize] = result;
```

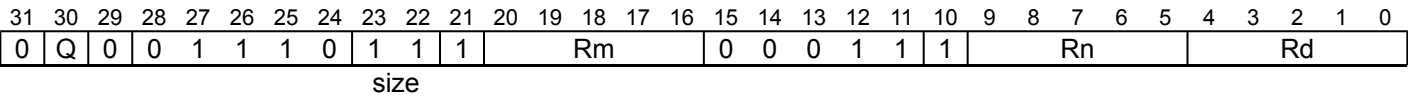
Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

ORN (vector)

Bitwise inclusive OR NOT (vector). This instruction performs a bitwise OR NOT between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



ORN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "Q":
- | Q | <T> |
|---|-----|
| 0 | 8B  |
| 1 | 16B |
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

operand2 = NOT(operand2);

result = operand1 OR operand2;

V[d, datasize] = result;
```

Operational information

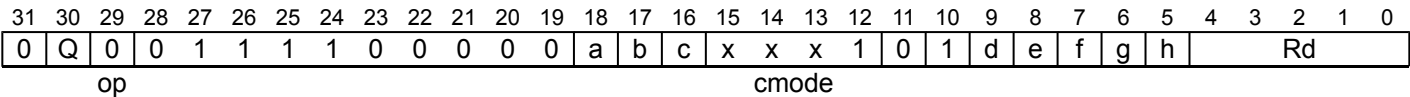
- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



ORR (vector, immediate)

Bitwise inclusive OR (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise OR between each result and an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



16-bit (cmode == 10x1)

```
ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit (cmode == 0xx1)

```
ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
  when '0xx00' operation = ImmediateOp_MOVI;
  when '0xx10' operation = ImmediateOp_ORR;
  when '10x00' operation = ImmediateOp_MOVI;
  when '10x10' operation = ImmediateOp_ORR;
  when '110x0' operation = ImmediateOp_MOVI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '11110' operation = ImmediateOp_MOVI;
imm64 = AdvSIMDEExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP register, encoded in the "Rd" field.

<T> For the 16-bit variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit variant: is the shift amount encoded in "cmode<1>":

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
  when ImmediateOp_MOVI
    result = imm;
  when ImmediateOp_MVNI
    result = NOT(imm);
  when ImmediateOp_ORR
    operand = V[rd, datasize];
    result = operand OR imm;
  when ImmediateOp_BIC
    operand = V[rd, datasize];
    result = operand AND NOT(imm);

V[rd, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

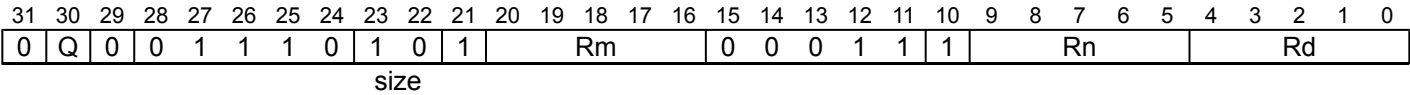
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORR (vector, register)

Bitwise inclusive OR (vector, register). This instruction performs a bitwise OR between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias [MOV \(vector\)](#).



ORR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "Q":
- | Q | <T> |
|---|-----|
| 0 | 8B  |
| 1 | 16B |
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Alias Conditions

Alias	Is preferred when
<a href="#">MOV (vector)</a>	Rm == Rn

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

result = operand1 OR operand2;

V[d, datasize] = result;
```

Operational information

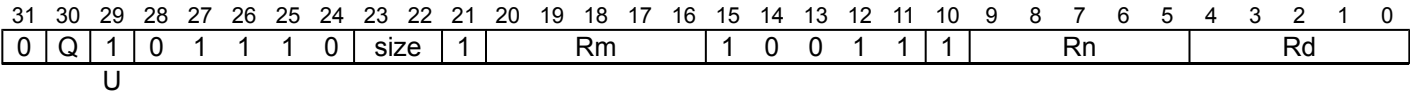
- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

PMUL

Polynomial Multiply. This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



PMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean poly = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
    else
        product = (UInt(element1)*UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PMULL, PMULL2

Polynomial Multiply Long. This instruction multiplies corresponding elements in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials, see *Polynomial arithmetic over {0, 1}*.

The PMULL instruction extracts each source vector from the lower half of each source register. The PMULL2 instruction extracts each source vector from the upper half of each source register.

The PMULL and PMULL2 variants that operate on 64-bit source elements are defined only when FEAT\_PMULL is implemented.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	Rm						1	1	1	0	0	0	Rn						Rd			

PMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '01' || size == '10' then UNDEFINED;
if size == '11' && !HaveBit128PMULLExt() then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	RESERVED
10	RESERVED
11	1Q

The '1Q' arrangement is only allocated in an implementation that includes the Cryptographic Extension, and is otherwise RESERVED.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	x	RESERVED
10	x	RESERVED
11	0	1D
11	1	2D

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, 2*esize] = PolynomialMult(element1, element2);

V[d, 2*datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

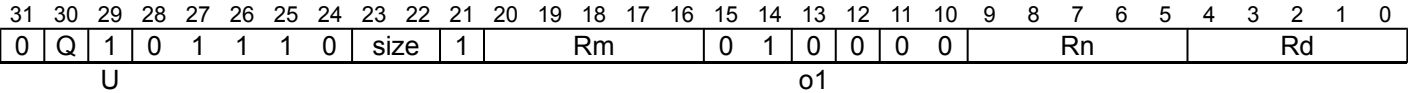
RADDHN, RADDHN2

Rounding Add returning High Narrow. This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see ADDHN.

The RADDHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RADDHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



RADDHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.



## Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(2*datasize) operand2 = V[m, 2*datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, 2*esize]);
    element2 = UInt(Elem[operand2, e, 2*esize]);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = RShr(sum, esize, round);
    Elem[result, e, esize] = sum<esize-1:0>;

Vpart[d, part, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

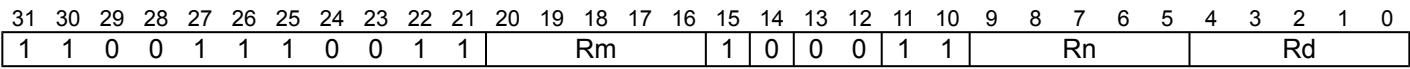
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RAX1

Rotate and Exclusive-OR rotates each 64-bit element of the 128-bit vector in a source SIMD&FP register left by 1, performs a bitwise exclusive-OR of the resulting 128-bit vector and the vector in another source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when **FEAT\_SHA3** is implemented.

Advanced SIMD  
(FEAT\_SHA3)



RAX1 <Vd>.2D, <Vn>.2D, <Vm>.2D

```
if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
V[d, 128] = Vn EOR (ROL(Vm<127:64>, 1):ROL(Vm<63:0>, 1));
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

RBIT (vector)

Reverse Bit order (vector). This instruction reads each vector element from the source SIMD&FP register, reverses the bits of the element, places the results into a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	Q	1	0	1	1	1	0	0	1	1	0	0	0	0	0	0	1	0	1	1	0	Rn						Rd					

RBIT <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;
bits(esize) rev;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    for i = 0 to esize-1
        rev<(esize-1)-i> = element<i>;
    Elem[result, e, esize] = rev;

V[d, datasize] = result;
```

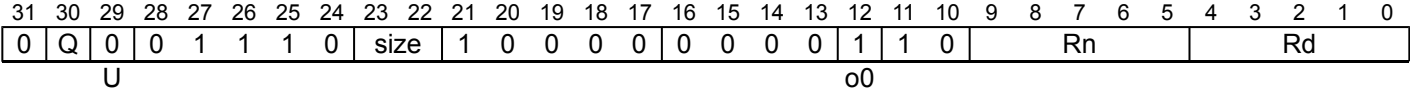
Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

REV16 (vector)

Reverse elements in 16-bit halfwords (vector). This instruction reverses the order of 8-bit elements in each halfword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



REV16 <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:  B(0),  H(1),  S(1),  D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx:  64(0),  32(1),  16(2)
bits(2) op = o0:U;

// => op+size:
//   64+B = 0,  64+H = 1,  64+S = 2,  64+D = X
//   32+B = 1,  32+H = 2,  32+S = X,  32+D = X
//   16+B = 2,  16+H = X,  16+S = X,  16+D = X
//   8+B = X,   8+H = X,   8+S = X,   8+D = X
// => 3-(op+size) (index bits in group)
//   64/B = 3,  64+H = 2,  64+S = 1,  64+D = X
//   32+B = 2,  32+H = 1,  32+S = X,  32+D = X
//   16+B = 1,  16+H = X,  16+S = X,  16+D = X
//   8+B = X,   8+H = X,   8+S = X,   8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = (element + elements_per_container) - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

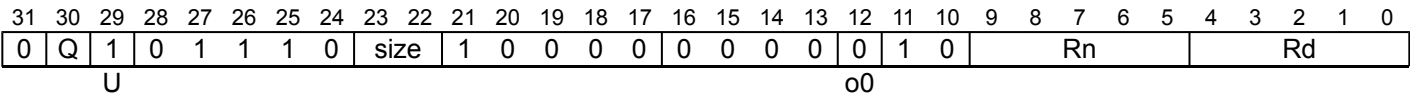
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

REV32 (vector)

Reverse elements in 32-bit words (vector). This instruction reverses the order of 8-bit or 16-bit elements in each word of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



REV32 <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:  B(0),  H(1),  S(1),  D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx:  64(0),  32(1),  16(2)
bits(2) op = o0:U;

// => op+size:
//   64+B = 0,  64+H = 1,  64+S = 2,  64+D = X
//   32+B = 1,  32+H = 2,  32+S = X,  32+D = X
//   16+B = 2,  16+H = X,  16+S = X,  16+D = X
//   8+B = X,   8+H = X,   8+S = X,   8+D = X
// => 3-(op+size) (index bits in group)
//   64/B = 3,  64+H = 2,  64+S = 1,  64+D = X
//   32+B = 2,  32+H = 1,  32+S = X,  32+D = X
//   16+B = 1,  16+H = X,  16+S = X,  16+D = X
//   8+B = X,   8+H = X,   8+S = X,   8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
1x	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = (element + elements_per_container) - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

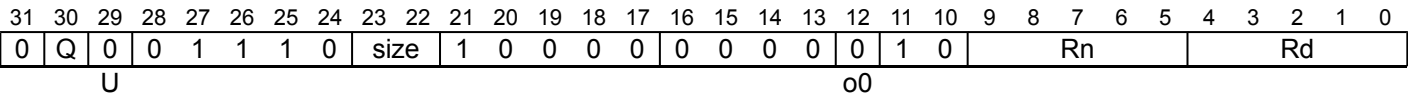
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

Reverse elements in 64-bit doublewords (vector). This instruction reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



REV64 <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:  B(0),  H(1),  S(1),  D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx:  64(0),  32(1),  16(2)
bits(2) op = o0:U;

// => op+size:
//   64+B = 0,  64+H = 1,  64+S = 2,  64+D = X
//   32+B = 1,  32+H = 2,  32+S = X,  32+D = X
//   16+B = 2,  16+H = X,  16+S = X,  16+D = X
//   8+B = X,   8+H = X,   8+S = X,   8+D = X
// => 3-(op+size) (index bits in group)
//   64/B = 3,  64+H = 2,  64+S = 1,  64+D = X
//   32+B = 2,  32+H = 1,  32+S = X,  32+D = X
//   16+B = 1,  16+H = X,  16+S = X,  16+D = X
//   8+B = X,   8+H = X,   8+S = X,   8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.



## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = (element + elements_per_container) - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RSHRN, RSHRN2

Rounding Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the vector in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see [SHRN](#).

The RSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb				1	0	0	0	1	1	Rn				Rd				
immh										op																					

RSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<shift> Is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immh”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immh))
001x	(32-UInt(immh:immh))
01xx	(64-UInt(immh:immh))
1xxx	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = RShr(UInt(Elem[operand, e, 2*esize]), shift, round);
    Elem[result, e, esize] = element<esize-1:0>;

Vpart[d, part, datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

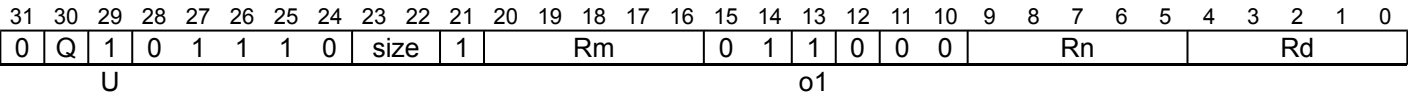
RSUBHN, RSUBHN2

Rounding Subtract returning High Narrow. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see SUBHN.

The RSUBHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSUBHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



RSUBHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(2*datasize) operand2 = V[m, 2*datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, 2*esize]);
    element2 = UInt(Elem[operand2, e, 2*esize]);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = RShr(sum, esize, round);
    Elem[result, e, esize] = sum<esize-1:0>;

Vpart[d, part, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

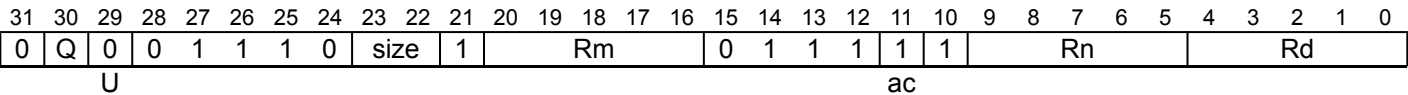
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SABA

Signed Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

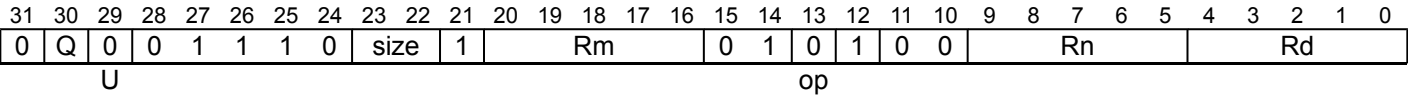
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SABAL, SABAL2

Signed Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABAL instruction extracts each source vector from the lower half of each source register. The SABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SABAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.



## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d, 2*datasize] else Zeros(2 * datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d, 2*datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

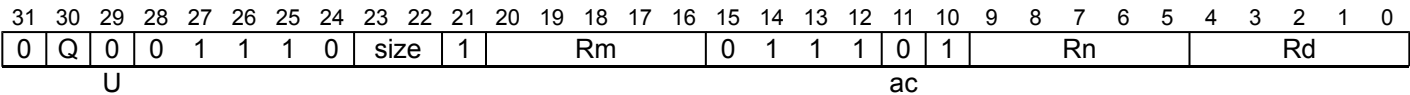
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SABD

Signed Absolute Difference. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

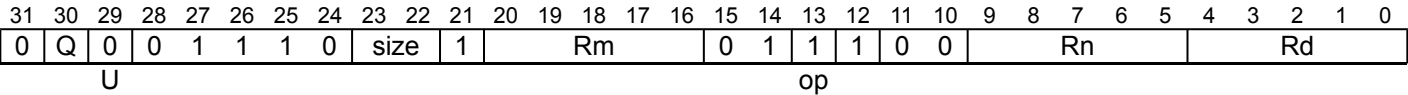
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SABDL, SABDL2

Signed Absolute Difference Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the results into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABDL instruction extracts each source vector from the lower half of each source register. The SABDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d, 2*datasize] else Zeros(2 * datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d, 2*datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

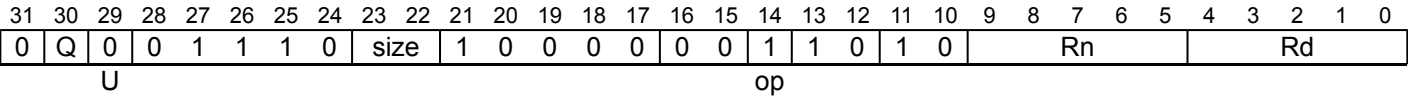
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SADALP

Signed Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register and accumulates the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SADALP <Vd>.<Ta>, <Vn>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d, datasize];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

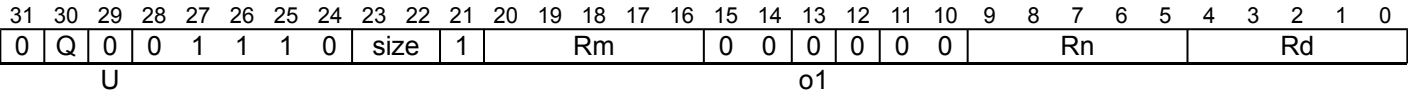
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SADDL, SADDL2

Signed Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SADDL instruction extracts each source vector from the lower half of each source register. The SADDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SADDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.



## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

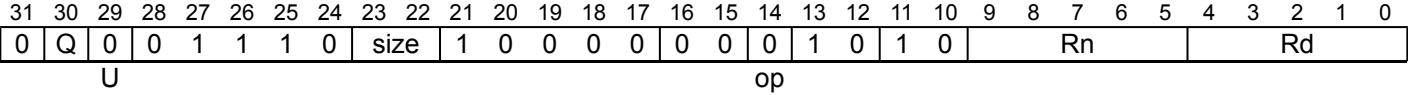
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SADDLP

Signed Add Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SADDLP <Vd>.<Ta>, <Vn>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d, datasize];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

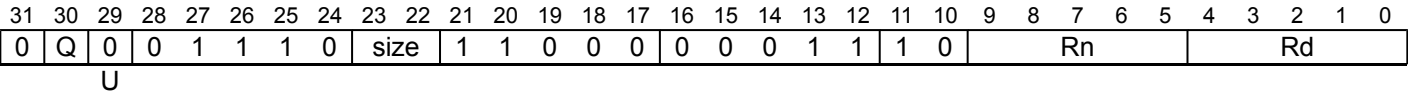
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SADDLV

Signed Add Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SADDLV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	H
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
integer sum;

sum = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    sum = sum + Int(Elem[operand, e, esize], unsigned);

V[d, 2*esize] = sum<2*esize-1:0>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

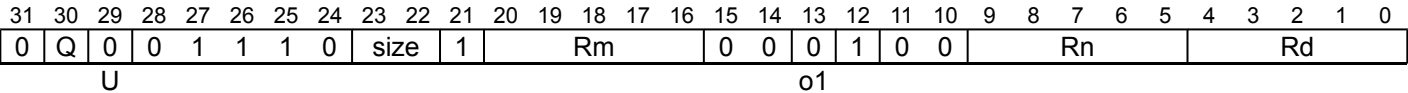
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SADDW, SADDW2

Signed Add Wide. This instruction adds vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the results in a vector, and writes the vector to the SIMD&FP destination register.

The SADDW instruction extracts the second source vector from the lower half of the second source register. The SADDW2 instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SADDW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SCVTF (scalar, fixed-point)

Signed fixed-point Convert to Floating-point (scalar). This instruction converts the signed value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the [FPCR](#), and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		0	0	1	1	1	1	0	ftype		0	0	0	0	1	0	scale					Rn					Rd				
rmode										opcode																					

### 32-bit to half-precision (sf == 0 && ftype == 11) (FEAT\_FP16)

SCVTF <Hd>, <Wn>, #<fbits>

### 32-bit to single-precision (sf == 0 && ftype == 00)

SCVTF <Sd>, <Wn>, #<fbits>

### 32-bit to double-precision (sf == 0 && ftype == 01)

SCVTF <Dd>, <Wn>, #<fbits>

### 64-bit to half-precision (sf == 1 && ftype == 11) (FEAT\_FP16)

SCVTF <Hd>, <Xn>, #<fbits>

### 64-bit to single-precision (sf == 1 && ftype == 00)

SCVTF <Sd>, <Xn>, #<fbits>

### 64-bit to double-precision (sf == 1 && ftype == 01)

SCVTF <Dd>, <Xn>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer decode_fltsize;
FPRounding rounding;

case ftype of
  when '00' decode_fltsize = 32;
  when '01' decode_fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      decode_fltsize = 16;
    else
      UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

rounding = FPRoundingMode(FPCR[]);
```



## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<fbits>	For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".  For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

## Operation

```
CheckFPEnabled64();  
  
FPCRTYPE fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
constant integer fltsize = if merge then 128 else decode_fltsize;  
bits(fltsize) fltval;  
bits(intsize) intval;  
  
intval = X[n, intsize];  
fltval = if merge then V[d, fltsize] else Zeros(fltsize);  
Elem[fltval, 0, decode_fltsize] = FixedToFP(intval, fracbits, FALSE, fpcr, rounding, decode_fltsize);  
V[d, fltsize] = fltval;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SCVTF (scalar, integer)

Signed integer Convert to Floating-point (scalar). This instruction converts the signed integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the [FPCR](#), and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	ftype	1	0	0	0	1	0	0	0	0	0	0	0	Rn				Rd						
rmode												opcode																			

### 32-bit to half-precision (sf == 0 && ftype == 11) (FEAT\_FP16)

SCVTF <Hd>, <Wn>

### 32-bit to single-precision (sf == 0 && ftype == 00)

SCVTF <Sd>, <Wn>

### 32-bit to double-precision (sf == 0 && ftype == 01)

SCVTF <Dd>, <Wn>

### 64-bit to half-precision (sf == 1 && ftype == 11) (FEAT\_FP16)

SCVTF <Hd>, <Xn>

### 64-bit to single-precision (sf == 1 && ftype == 00)

SCVTF <Sd>, <Xn>

### 64-bit to double-precision (sf == 1 && ftype == 01)

SCVTF <Dd>, <Xn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer decode_fltsize;
FPRounding rounding;

case ftype of
  when '00'
    decode_fltsize = 32;
  when '01'
    decode_fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      decode_fltsize = 16;
    else
      UNDEFINED;

rounding = FPRoundingMode(FPCR[]);
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRType fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
constant integer fltsize = if merge then 128 else decode_fltsize;  
bits(fltsize) fltval;  
bits(intsize) intval;  
  
intval = X[n, intsize];  
fltval = if merge then V[d, fltsize] else Zeros(fltsize);  
Elem[fltval, 0, decode_fltsize] = FixedToFP(intval, 0, FALSE, fpcr, rounding, decode_fltsize);  
V[d, fltsize] = fltval;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SCVTF (vector, fixed-point)

Signed fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the [FPCR](#), and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR EL1](#), [CPTR EL2](#), and [CPTR EL3](#) registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			1	1	1	0	0	1	Rn				Rd					
U									immh																						

SCVTF <V><d>, <V><n>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR[]);
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			1	1	1	0	0	1	Rn				Rd					
U									immh																						

SCVTF <Vd>.<T>, <Vn>.<T>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR[]);
```

### Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
000x	RESERVED
001x	H
01xx	S
1xxx	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	x	RESERVED
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in "immh:immb":

immh	<fbits>
000x	RESERVED
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in "immh:immb":

immh	<fbits>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	RESERVED
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, fpcr, rounding, esize);
V[d, 128] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SCVTF (vector, integer)

Signed integer Convert to Floating-point (vector). This instruction converts each element in a vector from signed integer to floating-point using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

Scalar half precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn				Rd					
U																															

SCVTF <Hd> , <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn				Rd					
U																															

SCVTF <V><d> , <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector half precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn				Rd					
U																															

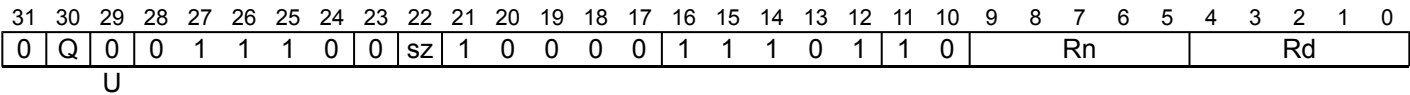
SCVTF <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



SCVTF <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

FPRounding rounding = FPRoundingMode(fpcr);
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



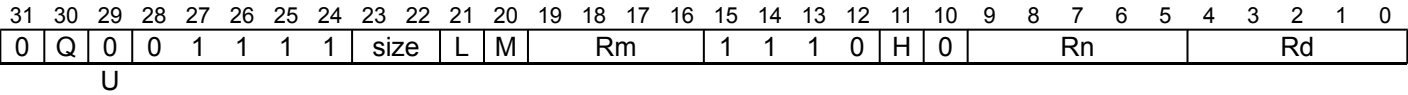
SDOT (by element)

Dot Product signed arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register. Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

*ID\_AA64ISAR0\_EL1*.DP indicates whether this instruction is supported.

Vector  
(FEAT\_DotProd)



SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.**4B**[<index>]

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index = UInt(H:L);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "Q":

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<index> Is the element index, encoded in the "H:L" fields.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(128) operand2 = V[m, 128];
bits(datasize) result = V[d, datasize];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*index+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*index+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d, datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SDOT (vector)

Dot Product signed arithmetic (vector). This instruction performs the dot product of the four signed 8-bit elements in each 32-bit element of the first source register with the four signed 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

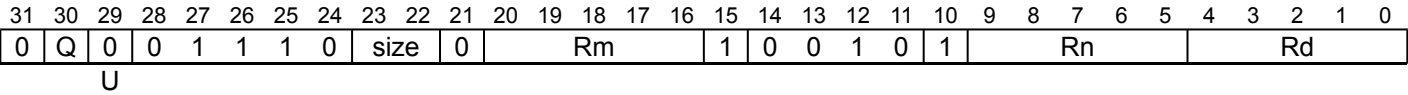
Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

[ID\\_AA64ISAR0\\_EL1](#).DP indicates whether this instruction is supported.

Vector  
(FEAT\_DotProd)



SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "Q":

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

result = V[d, datasize];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d, datasize] = result;
```

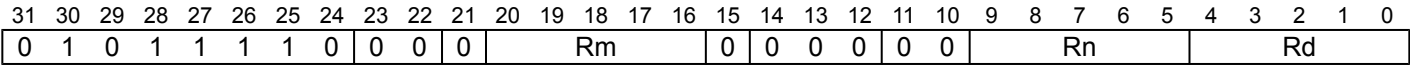
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA1C

SHA1 hash update (choose).

Advanced SIMD  
(FEAT\_SHA1)



SHA1C <Qd>, <Sn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) x = V[d, 128];
bits(32) y = V[n, 32]; // Note: 32 not 128 bits wide
bits(128) w = V[m, 128];
bits(32) t;

for e = 0 to 3
    t = SHAchoose(x<63:32>, x<95:64>, x<127:96>);
    y = y + ROL(x<31:0>, 5) + t + Elem[w, e, 32];
    x<63:32> = ROL(x<63:32>, 30);
    <y, x> = ROL(y:x, 32);
V[d, 128] = x;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

# SHA1H

SHA1 fixed rotate.

## Advanced SIMD (FEAT\_SHA1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	Rn				Rd					

SHA1H <Sd>, <Sn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA1Ext() then UNDEFINED;
```

## Assembler Symbols

- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(32) operand = V[n, 32]; // read element [0] only, [1-3] zeroed
V[d, 32] = ROL(operand, 30);
```

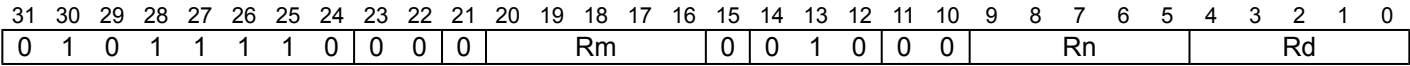
## Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

SHA1M

SHA1 hash update (majority).

Advanced SIMD  
(FEAT\_SHA1)



SHA1M <Qd>, <Sn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) x = V[d, 128];
bits(32) y = V[n, 32]; // Note: 32 not 128 bits wide
bits(128) w = V[m, 128];
bits(32) t;

for e = 0 to 3
    t = SHAmajority(x<63:32>, x<95:64>, x<127:96>);
    y = y + ROL(x<31:0>, 5) + t + Elem[w, e, 32];
    x<63:32> = ROL(x<63:32>, 30);
    <y, x> = ROL(y:x, 32);
V[d, 128] = x;
```

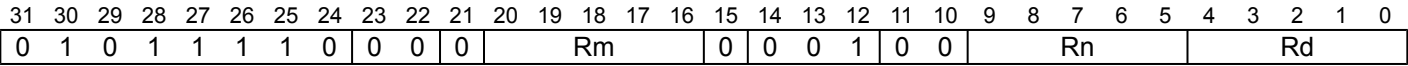
Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

# SHA1P

SHA1 hash update (parity).

## Advanced SIMD (FEAT\_SHA1)



SHA1P <Qd>, <Sn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

### Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) x = V[d, 128];
bits(32) y = V[n, 32]; // Note: 32 not 128 bits wide
bits(128) w = V[m, 128];
bits(32) t;

for e = 0 to 3
    t = SHAParity(x<63:32>, x<95:64>, x<127:96>);
    y = y + ROL(x<31:0>, 5) + t + Elem[w, e, 32];
    x<63:32> = ROL(x<63:32>, 30);
    <y, x> = ROL(y:x, 32);
V[d, 128] = x;
```

### Operational information

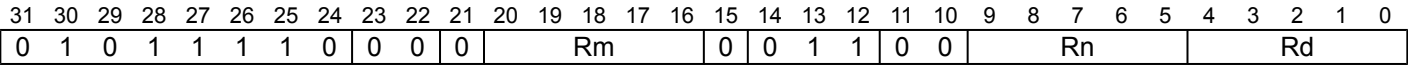
- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



# SHA1SU0

SHA1 schedule update 0.

## Advanced SIMD (FEAT\_SHA1)



SHA1SU0 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

## Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d, 128];
bits(128) operand2 = V[n, 128];
bits(128) operand3 = V[m, 128];
bits(128) result;

result = operand2<63:0>:operand1<127:64>;
result = result EOR operand1 EOR operand3;
V[d, 128] = result;
```

## Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

# SHA1SU1

SHA1 schedule update 1.

## Advanced SIMD (FEAT\_SHA1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	0	1	0	0	0	0	0	0	1	1	0	Rn				Rd					

SHA1SU1 <Vd>.4S, <Vn>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA1Ext() then UNDEFINED;
```

## Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d, 128];
bits(128) operand2 = V[n, 128];
bits(128) result;
bits(128) T = operand1 EOR LSR(operand2, 32);
result<31:0> = ROL(T<31:0>, 1);
result<63:32> = ROL(T<63:32>, 1);
result<95:64> = ROL(T<95:64>, 1);
result<127:96> = ROL(T<127:96>, 1) EOR ROL(T<31:0>, 2);
V[d, 128] = result;
```

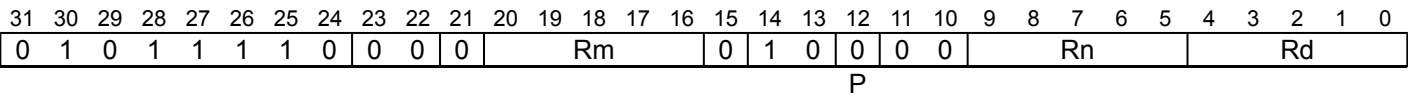
## Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

SHA256H

SHA256 hash update (part 1).

Advanced SIMD  
(FEAT\_SHA256)



SHA256H <Qd>, <Qn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;
```

Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- <Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) result;
result = SHA256hash(V[d, 128], V[n, 128], V[m, 128], TRUE);
V[d, 128] = result;
```

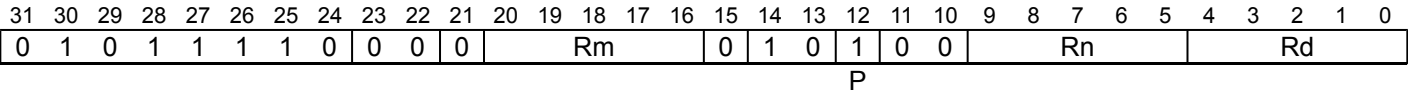
Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

# SHA256H2

SHA256 hash update (part 2).

## Advanced SIMD (FEAT\_SHA256)



SHA256H2 <Qd>, <Qn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;
```

### Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- <Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) result;
result = SHA256hash(V[n, 128], V[d, 128], V[m, 128], FALSE);
V[d, 128] = result;
```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

SHA256SU0

SHA256 schedule update 0.

Advanced SIMD  
(FEAT\_SHA256)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	Rn				Rd					

SHA256SU0 <Vd>.4S, <Vn>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA256Ext() then UNDEFINED;
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d, 128];
bits(128) operand2 = V[n, 128];
bits(128) result;
bits(128) T = operand2<31:0>:operand1<127:32>;
bits(32) elt;

for e = 0 to 3
    elt = Elem[T, e, 32];
    elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
    Elem[result, e, 32] = elt + Elem[operand1, e, 32];
V[d, 128] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

# SHA256SU1

SHA256 schedule update 1.

## Advanced SIMD (FEAT\_SHA256)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	0	0	Rm					0	1	1	0	0	0	Rn					Rd				

SHA256SU1 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;
```

## Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d, 128];
bits(128) operand2 = V[n, 128];
bits(128) operand3 = V[m, 128];
bits(128) result;
bits(128) T0 = operand3<31:0>:operand2<127:32>;
bits(64) T1;
bits(32) elt;

T1 = operand3<127:64>;
for e = 0 to 1
    elt = Elem[T1, e, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

T1 = result<63:0>;
for e = 2 to 3
    elt = Elem[T1, e-2, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

V[d, 128] = result;
```

## Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



SHA512H

SHA512 Hash update part 1 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma1 and chi functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when *FEAT\_SHA512* is implemented.

Advanced SIMD  
(FEAT\_SHA512)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1	Rm					1	0	0	0	0	0	Rn					Rd				

SHA512H <Qd>, <Qn>, <Vm>.2D

```
if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vtmp;
bits(64) MSigma1;
bits(64) tmp;
bits(128) x = V[n, 128];
bits(128) y = V[m, 128];
bits(128) w = V[d, 128];

MSigma1 = ROR(y<127:64>, 14) EOR ROR(y<127:64>, 18) EOR ROR(y<127:64>, 41);
Vtmp<127:64> = (y<127:64> AND x<63:0>) EOR (NOT(y<127:64>) AND x<127:64>);
Vtmp<127:64> = (Vtmp<127:64> + MSigma1 + w<127:64>);
tmp = Vtmp<127:64> + y<63:0>;
MSigma1 = ROR(tmp, 14) EOR ROR(tmp, 18) EOR ROR(tmp, 41);
Vtmp<63:0> = (tmp AND y<127:64>) EOR (NOT(tmp) AND x<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + MSigma1 + w<63:0>);
V[d, 128] = Vtmp;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



SHA512H2

SHA512 Hash update part 2 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma0 and majority functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when *FEAT\_SHA512* is implemented.

Advanced SIMD  
(FEAT\_SHA512)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1	Rm				1	0	0	0	0	1	Rn				Rd						

SHA512H2 <Qd>, <Qn>, <Vm>.2D

```
if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

- <Qd> Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vtmp;
bits(64) NSigma0;
bits(128) x = V[n, 128];
bits(128) y = V[m, 128];
bits(128) w = V[d, 128];

NSigma0 = ROR(y<63:0>, 28) EOR ROR(y<63:0>, 34) EOR ROR(y<63:0>, 39);
Vtmp<127:64> = (x<63:0> AND y<127:64>) EOR (x<63:0> AND y<63:0>) EOR (y<127:64> AND y<63:0>);
Vtmp<127:64> = (Vtmp<127:64> + NSigma0 + w<127:64>);
NSigma0 = ROR(Vtmp<127:64>, 28) EOR ROR(Vtmp<127:64>, 34) EOR ROR(Vtmp<127:64>, 39);
Vtmp<63:0> = ((Vtmp<127:64> AND y<63:0>) EOR (Vtmp<127:64> AND y<127:64>) EOR (y<127:64> AND y<63:0>));
Vtmp<63:0> = (Vtmp<63:0> + NSigma0 + w<63:0>);

V[d, 128] = Vtmp;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

SHA512SU0

SHA512 Schedule Update 0 takes the values from the two 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the gamma0 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when *FEAT\_SHA512* is implemented.

Advanced SIMD  
(FEAT\_SHA512)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	Rn				Rd				

SHA512SU0 <Vd>.2D, <Vn>.2D

```
if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) sig0;
bits(128) Vtmp;
bits(128) x = V[n, 128];
bits(128) w = V[d, 128];
sig0 = ROR(w<127:64>, 1) EOR ROR(w<127:64>, 8) EOR ('0000000':w<127:71>);
Vtmp<63:0> = w<63:0> + sig0;
sig0 = ROR(x<63:0>, 1) EOR ROR(x<63:0>, 8) EOR ('0000000':x<63:7>);
Vtmp<127:64> = w<127:64> + sig0;
V[d, 128] = Vtmp;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

SHA512SU1

SHA512 Schedule Update 1 takes the values from the three source SIMD&FP registers and produces a 128-bit output value that combines the gamma1 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when *FEAT\_SHA512* is implemented.

Advanced SIMD  
(FEAT\_SHA512)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1	Rm				1	0	0	0	1	0	Rn				Rd						

SHA512SU1 <Vd>.2D, <Vn>.2D, <Vm>.2D

```
if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) sig1;
bits(128) Vtmp;
bits(128) x = V[n, 128];
bits(128) y = V[m, 128];
bits(128) w = V[d, 128];

sig1 = ROR(x<127:64>, 19) EOR ROR(x<127:64>, 61) EOR ('000000':x<127:70>);
Vtmp<127:64> = w<127:64> + sig1 + y<127:64>;
sig1 = ROR(x<63:0>, 19) EOR ROR(x<63:0>, 61) EOR ('000000':x<63:6>);
Vtmp<63:0> = w<63:0> + sig1 + y<63:0>;
V[d, 128] = Vtmp;
```

Operational information

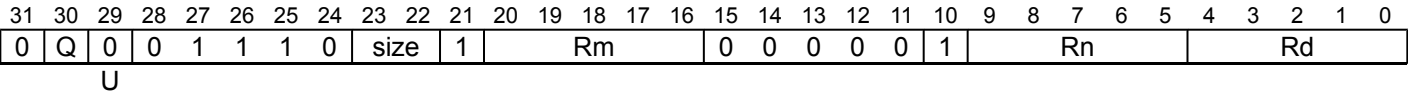
- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

SHADD

Signed Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see [SRHADD](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = (element1 + element2) >> 1;
    Elem[result, e, esize] = sum<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

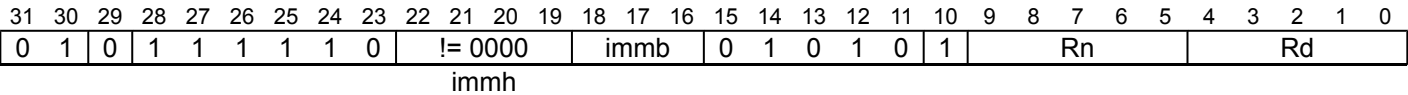
SHL

Shift Left (immediate). This instruction reads each value from a vector, left shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar



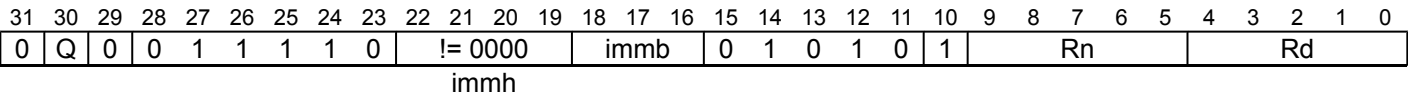
```
SHL <V><d>, <V><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;
```

Vector



```
SHL <Vd>.<T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
```

Assembler Symbols

- <V> Is a width specifier, encoded in “immh”:
- | immh |  | <V>      |
|------|--|----------|
| 0xxx |  | RESERVED |
| 1xxx |  | D        |
- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in "immh:immb":

immh	<shift>
0xxx	RESERVED
1xxx	(UInt(immh:immb)-64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in "immh:immb":

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = LSL(Elem[operand, e, esize], shift);
V[d, datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHLL, SHLL2

Shift Left Long (by element size). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, left shifts each result by the element size, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. The SHLL instruction extracts vector elements from the lower half of the source register. The SHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	0	0	0	0	1	0	0	1	1	1	0	Rn				Rd					

SHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = esize;
boolean unsigned = FALSE;    // Or TRUE without change of functionality
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<shift> Is the left shift amount, which must be equal to the source element width in bits, encoded in “size”:



size	<shift>
00	8
01	16
10	32
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part, datasize];
bits(2*datasize) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d, 2*datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

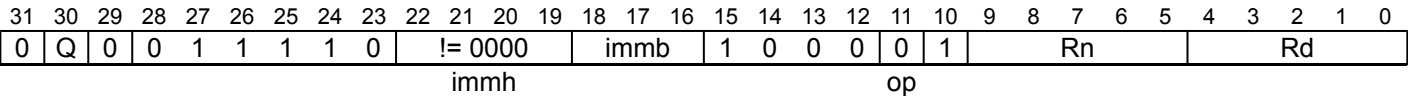
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHRN, SHRN2

Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are truncated. For rounded results, see [RSHRN](#).

The RSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



```
SHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<shift> Is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immh”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immh))
001x	(32-UInt(immh:immh))
01xx	(64-UInt(immh:immh))
1xxx	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = RShr(UInt(Elem[operand, e, 2*esize]), shift, round);
    Elem[result, e, esize] = element<esize-1:0>;

Vpart[d, part, datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

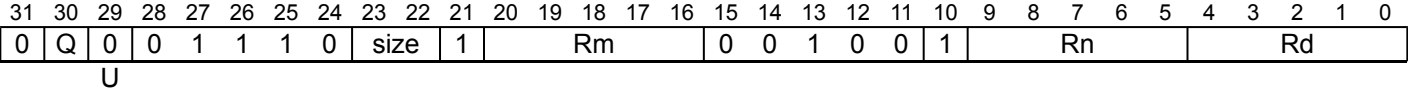
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHSUB

Signed Halving Subtract. This instruction subtracts the elements in the vector in the second source SIMD&FP register from the corresponding elements in the vector in the first source SIMD&FP register, shifts each result right one bit, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



```
SHSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer diff;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = (element1 - element2) >> 1;
    Elem[result, e, esize] = diff<esize-1:0>;

V[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

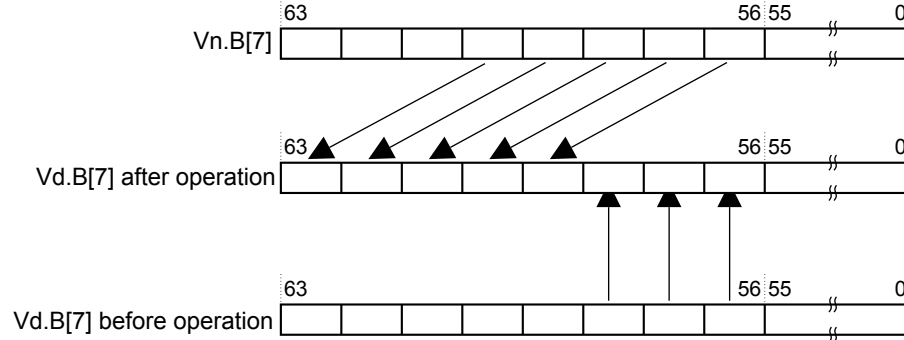
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SLI

Shift Left and Insert (immediate). This instruction reads each vector element in the source SIMD&FP register, left shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the left of each vector element in the source register are lost.

The following figure shows an example of the operation of shift left by 3 for an 8-bit vector element.



Depending on the settings in the [CPACR EL1](#), [CPTR EL2](#), and [CPTR EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000			immb			0	1	0	1	0	1	1	Rn				Rd					
immh																															

**SLI** [<V><d>](#), [<V><n>](#), [#<shift>](#)

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	1	0	1	0	1	Rn				Rd					
immh																															

**SLI** [<Vd>.<T>](#), [<Vn>.<T>](#), [#<shift>](#)

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(UInt(immh:immb)-64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2 = V[d, datasize];
bits(datasize) result;
bits(esize) mask = LSL(Ones(esize), shift);
bits(esize) shifted;

for e = 0 to elements-1
    shifted = LSL(Elem[operand, e, esize], shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SM3PARTW1

SM3PARTW1 takes three 128-bit vectors from the three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive-OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when *FEAT\_SM3* is implemented.

Advanced SIMD  
(FEAT\_SM3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1	Rm				1	1	0	0	0	0	Rn				Rd						

SM3PARTW1 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Vd = V[d, 128];
bits(128) result;

result<95:0> = (Vd EOR Vn)<95:0> EOR (ROL(Vm<127:96>, 15):ROL(Vm<95:64>, 15):ROL(Vm<63:32>, 15));

for i = 0 to 3
    if i == 3 then
        result<127:96> = (Vd EOR Vn)<127:96> EOR (ROL(result<31:0>, 15));
        result<(32*i)+31:(32*i)> = (result<(32*i)+31:(32*i)> EOR ROL(result<(32*i)+31:(32*i)>, 15) EOR ROL(re
V[d, 128] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

# SM3PARTW2

SM3PARTW2 takes three 128-bit vectors from three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive-OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when **FEAT\_SM3** is implemented.

## Advanced SIMD (FEAT\_SM3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1	Rm				1	1	0	0	0	1	Rn				Rd						

SM3PARTW2 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

## Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Vd = V[d, 128];
bits(128) result;
bits(128) tmp;
bits(32) tmp2;
tmp<127:0> = Vn EOR (ROL(Vm<127:96>, 7):ROL(Vm<95:64>, 7):ROL(Vm<63:32>, 7):ROL(Vm<31:0>, 7));
result<127:0> = Vd<127:0> EOR tmp<127:0>;
tmp2 = ROL(tmp<31:0>, 15);
tmp2 = tmp2 EOR ROL(tmp2, 15) EOR ROL(tmp2, 23);
result<127:96> = result<127:96> EOR tmp2;
V[d, 128] = result;
```

## Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

SM3SS1

SM3SS1 rotates the top 32 bits of the 128-bit vector in the first source SIMD&FP register by 12, and adds that 32-bit value to the two other 32-bit values held in the top 32 bits of each of the 128-bit vectors in the second and third source SIMD&FP registers, rotating this result left by 7 and writing the final result into the top 32 bits of the vector in the destination SIMD&FP register, with the bottom 96 bits of the vector being written to 0.

This instruction is implemented only when *FEAT\_SM3* is implemented.

Advanced SIMD  
(FEAT\_SM3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	0	Rm				0	Ra				Rn				Rd							

SM3SS1 <Vd>.4S, <Vn>.4S, <Vm>.4S, <Va>.4S

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- <Va> Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Va = V[a, 128];
bits(128) result;
result<127:96> = ROL((ROL(Vn<127:96>, 12) + Vm<127:96> + Va<127:96>), 7);
result<95:0> = Zeros(96);
V[d, 128] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

# SM3TT1A

SM3TT1A takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive-OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive-OR.
- The result of the exclusive-OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

This instruction is implemented only when **FEAT\_SM3** is implemented.

## Advanced SIMD

(FEAT\_SM3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	0	Rm				1	0	imm2	0	0	Rn				Rd							

SM3TT1A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);
```

## Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
- <imm2> Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

## Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Vd = V[d, 128];
bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;

WjPrime = Elem[Vm, i, 32];
SS2 = Vn<127:96> EOR ROL(Vd<127:96>, 12);
TT1 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT1 = (TT1+Vd<31:0>+SS2+WjPrime)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 9);
result<95:64> = Vd<127:96>;
result<127:96> = TT1;
V[d, 128] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SM3TT1B

SM3TT1B takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The result of the exclusive-OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9. This instruction is implemented only when **FEAT\_SM3** is implemented.

Advanced SIMD

(FEAT\_SM3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	0	Rm				1	0	imm2	0	1	Rn				Rd							

SM3TT1B <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
- <imm2> Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Vd = V[d, 128];
bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;

WjPrime = Elem[Vm, i, 32];
SS2 = Vn<127:96> EOR ROL(Vd<127:96>, 12);
TT1 = (Vd<127:96> AND Vd<63:32>) OR (Vd<127:96> AND Vd<95:64>) OR (Vd<63:32> AND Vd<95:64>);
TT1 = (TT1+Vd<31:0>+SS2+WjPrime)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 9);
result<95:64> = Vd<127:96>;
result<127:96> = TT1;
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SM3TT2A

SM3TT2A takes three 128-bit vectors from three source SIMD&FP register and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive-OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive-OR.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive-OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive-OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when **FEAT\_SM3** is implemented.

Advanced SIMD  
(FEAT\_SM3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	0	Rm				1	0	imm2		1	0	Rn				Rd						

SM3TT2A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
- <imm2> Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Vd = V[d, 128];
bits(32) Wj;
bits(128) result;
bits(32) TT2;

Wj = Elem[Vm, i, 32];
TT2 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT2 = (TT2+Vd<31:0>+Vn<127:96>+Wj)<31:0>;

result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR ROL(TT2, 17);
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:



- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SM3TT2B

SM3TT2B takes three 128-bit vectors from three source SIMD&FP registers, and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive-OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive-OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when **FEAT\_SM3** is implemented.

Advanced SIMD  
(FEAT\_SM3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	0	Rm					1	0	imm2		1	1	Rn					Rd				

SM3TT2B <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
- <imm2> Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Vd = V[d, 128];
bits(32) Wj;
bits(128) result;
bits(32) TT2;

Wj = Elem[Vm, i, 32];
TT2 = (Vd<127:96> AND Vd<95:64>) OR (NOT(Vd<127:96>) AND Vd<63:32>);
TT2 = (TT2+Vd<31:0>+Vn<127:96>+Wj)<31:0>;

result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR ROL(TT2, 17);
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SM4E

SM4 Encode takes input data as a 128-bit vector from the first source SIMD&FP register, and four iterations of the round key held as the elements of the 128-bit vector in the second source SIMD&FP register. It encrypts the data by four rounds, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register. This instruction is implemented only when *FEAT\_SM4* is implemented.

Advanced SIMD  
(FEAT\_SM4)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	1	1	0	0	0	0	0	0	1	0	0	0	0	1										

SM4E <Vd>.4S, <Vn>.4S

```
if !HaveSM4Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vn = V[n, 128];
bits(32) intval;
bits(128) roundresult;
bits(32) roundkey;

roundresult = V[d, 128];
for index = 0 to 3
    roundkey = Elem[Vn, index, 32];

    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;

    for i = 0 to 3
        Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);

    intval = intval EOR ROL(intval, 2) EOR ROL(intval, 10) EOR ROL(intval, 18) EOR ROL(intval, 24);
    intval = intval EOR roundresult<31:0>;

    roundresult<31:0> = roundresult<63:32>;
    roundresult<63:32> = roundresult<95:64>;
    roundresult<95:64> = roundresult<127:96>;
    roundresult<127:96> = intval;

V[d, 128] = roundresult;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



SM4EKEY

SM4 Key takes an input as a 128-bit vector from the first source SIMD&FP register and a 128-bit constant from the second SIMD&FP register. It derives four iterations of the output key, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

This instruction is implemented only when *FEAT\_SM4* is implemented.

Advanced SIMD  
(FEAT\_SM4)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1	Rm				1	1	0	0	1	0	Rn				Rd						

SM4EKEY <Vd>.4S, <Vn>.4S, <Vm>.4S

```
if !HaveSM4Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(32) intval;
bits(32) const;
bits(128) roundresult;

roundresult = V[n, 128];
for index = 0 to 3
    const = Elem[Vm, index, 32];

    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;

    for i = 0 to 3
        Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);

    intval = intval EOR ROL(intval, 13) EOR ROL(intval, 23);
    intval = intval EOR roundresult<31:0>;

    roundresult<31:0> = roundresult<63:32>;
    roundresult<63:32> = roundresult<95:64>;
    roundresult<95:64> = roundresult<127:96>;
    roundresult<127:96> = intval;

V[d, 128] = roundresult;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

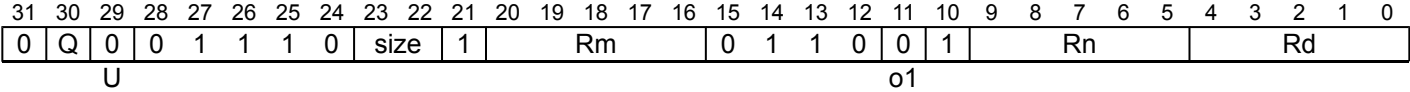
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMAX

Signed Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;
```



## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

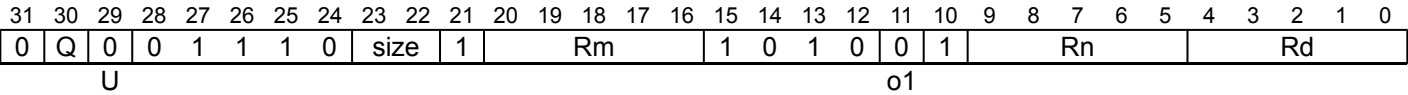
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMAXP

Signed Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

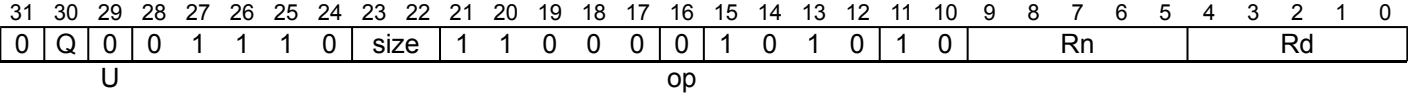
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMAXV

Signed Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMAXV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d, esize] = maxmin<esize-1:0>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

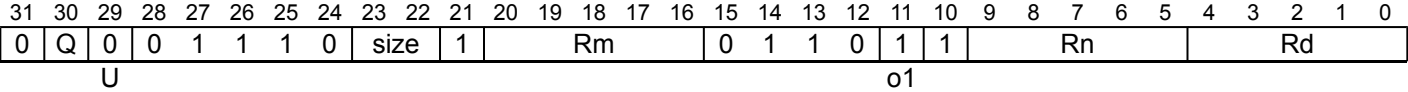
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMIN

Signed Minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

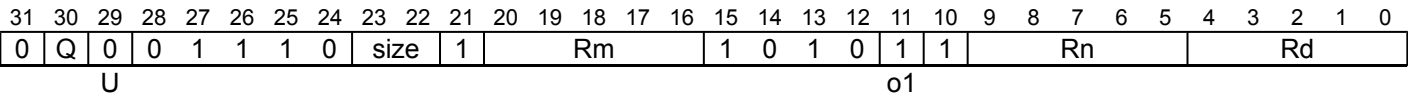
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMINP

Signed Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;
```



## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

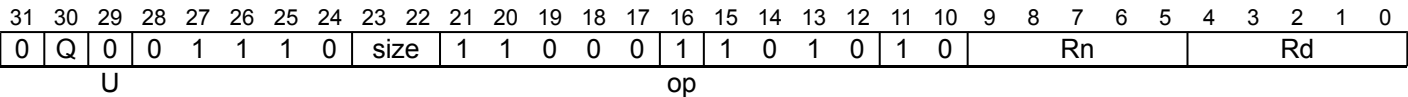
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMINV

Signed Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMINV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d, esize] = maxmin<esize-1:0>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

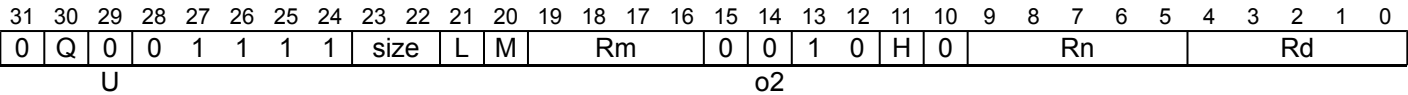
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLAL, SMLAL2 (by element)

Signed Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

The SMLAL instruction extracts vector elements from the lower half of the first source register. The SMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR EL1, CPTR EL2, and CPTR EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxsize) operand2 = V[m, idxsize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d, 2*datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

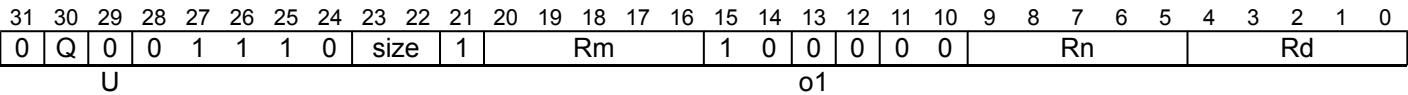


SMLAL, SMLAL2 (vector)

Signed Multiply-Add Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLAL instruction extracts each source vector from the lower half of each source register. The SMLAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR EL1, CPTR EL2, and CPTR EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d, 2*datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

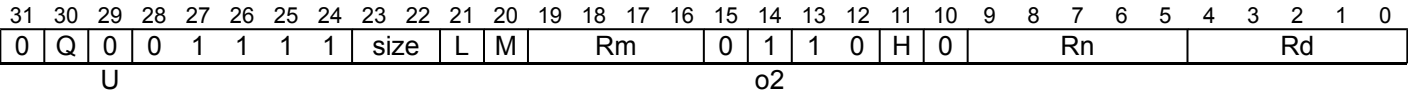


SMLS, SMLS2 (by element)

Signed Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLS instruction extracts vector elements from the lower half of the first source register. The SMLS2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR EL1, CPTR EL2, and CPTR EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMLS{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxsize) operand2 = V[m, idxsize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d, 2*datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

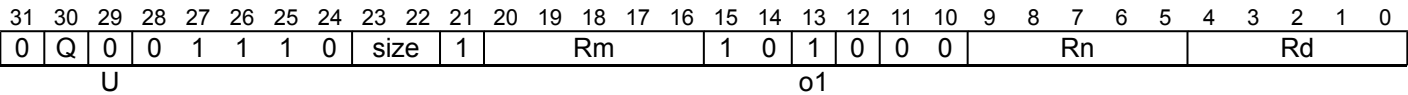


SMLSL, SMLSL2 (vector)

Signed Multiply-Subtract Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLSL instruction extracts each source vector from the lower half of each source register. The SMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR EL1, CPTR EL2, and CPTR EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d, 2*datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

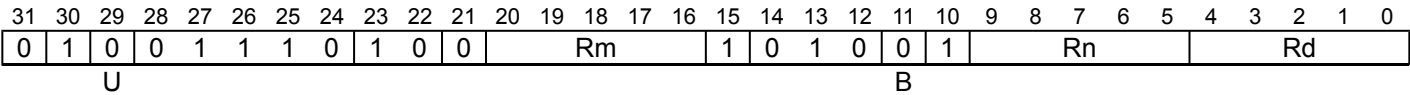
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMMLA (vector)

Signed 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of signed 8-bit integer values in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. [ID\\_AA64ISAR1\\_EL1](#).I8MM indicates whether this instruction is supported.

Vector  
(FEAT\_I8MM)



SMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B

```
if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n, 128];
bits(128) operand2 = V[m, 128];
bits(128) addend = V[d, 128];

V[d, 128] = MatMulAdd(addend, operand1, operand2, FALSE, FALSE);
```

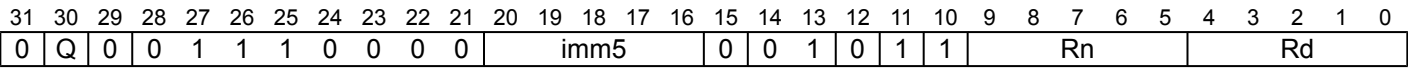
Operational information

Arm expects that the SMMLA (vector) instruction will deliver a peak integer multiply throughput that is at least as high as can be achieved using two SDOT (vector) instructions, with a goal that it should have significantly higher throughput.

SMOV

Signed Move vector element to general-purpose register. This instruction reads the signed integer from the source SIMD&FP register, sign-extends it to form a 32-bit or 64-bit value, and writes the result to destination general-purpose register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



32-bit (Q == 0)

```
SMOV <Wd>, <Vn>.<Ts>[<index>]
```

64-bit (Q == 1)

```
SMOV <Xd>, <Vn>.<Ts>[<index>]
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
    when 'xxxxx1' size = 0;    // SMOV [WX]d, Vn.B
    when 'xxxx10' size = 1;    // SMOV [WX]d, Vn.H
    when '1xx100' size = 2;    // SMOV Xd, Vn.S
    otherwise UNDEFINED;

integer idxdsize = if imm5<4> == '1' then 128 else 64;
integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <Ts> For the 32-bit variant: is an element size specifier, encoded in "imm5":

imm5	<Ts>
xxx00	RESERVED
xxxx1	B
xxx10	H

For the 64-bit variant: is an element size specifier, encoded in "imm5":

imm5	<Ts>
xx000	RESERVED
xxxx1	B
xxx10	H
xx100	S

<index> For the 32-bit variant: is the element index encoded in "imm5":

imm5	<index>
xxx00	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>

For the 64-bit variant: is the element index encoded in “imm5”:

imm5	<index>
xx000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>

## Operation

```
if index == 0 then
    CheckFPEnabled64();
else
    CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n, idxdsize];

X[d, datasize] = SignExtend(Elem[operand, index, esize], datasize);
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

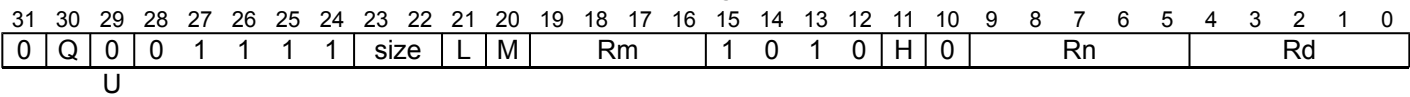


SMULL, SMULL2 (by element)

Signed Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts vector elements from the lower half of the first source register. The SMULL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxsize) operand2 = V[m, idxsize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d, 2*datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

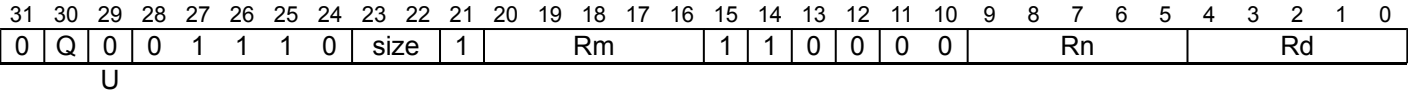
SMULL, SMULL2 (vector)

Signed Multiply Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts each source vector from the lower half of each source register. The SMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1*element2)<2*esize-1:0>;

V[d, 2*datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQABS

Signed saturating Absolute value. This instruction reads each vector element from the source SIMD&FP register, puts the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR.QC* is set.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Scalar* and *Vector*

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	1	0	0	0	0	0	0	0	1	1	1	1	0	Rn					Rd				
U																															

SQABS <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	0	0	0	1	1	1	1	0	Rn					Rd				
U																															

SQABS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

- <V>

Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D
- <d>

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n>

Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T>

Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

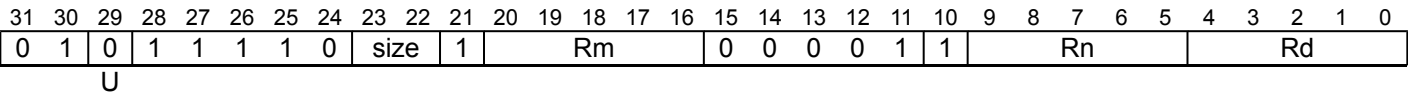
SQADD

Signed saturating Add. This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR.QC* is set.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

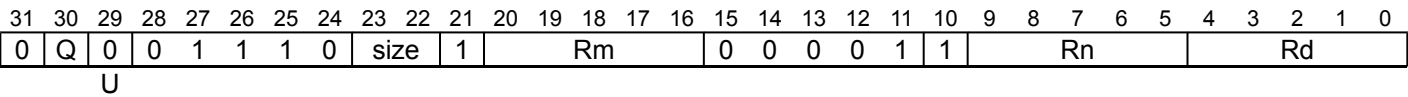
Scalar



SQADD <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector



SQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```



SQDMLAL, SQDMLAL2 (by element)

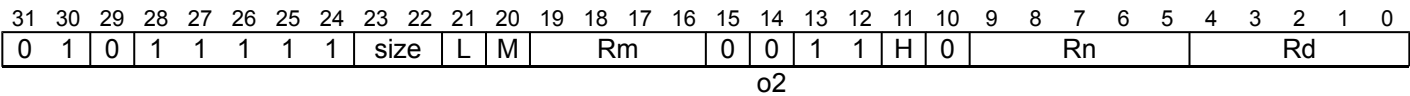
Signed saturating Doubling Multiply-Add Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR.QC* is set.

The *SQDMLAL* instruction extracts vector elements from the lower half of the first source register. The *SQDMLAL2* instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Scalar* and *Vector*

Scalar



SQDMLAL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]

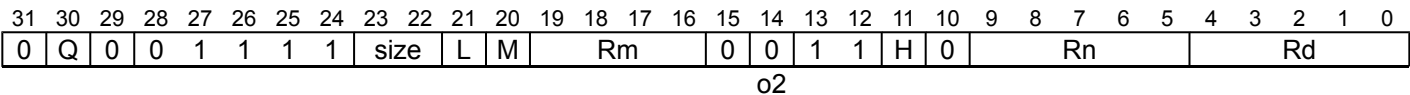
```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
```

Vector



SQDMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in "size:L:H:M":

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d, 2*datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMLAL, SQDMLAL2 (vector)

Signed saturating Doubling Multiply-Add Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

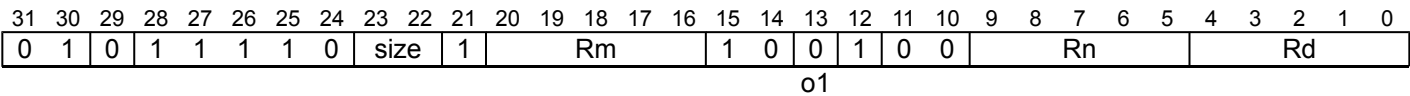
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR.QC* is set.

The *SQDMLAL* instruction extracts each source vector from the lower half of each source register. The *SQDMLAL2* instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Scalar* and *Vector*

Scalar



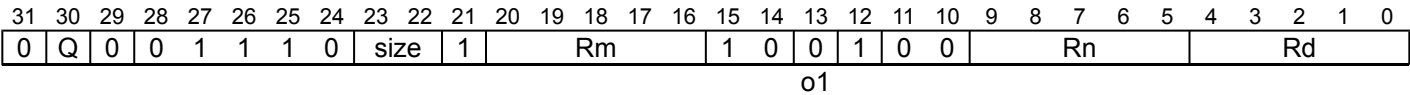
SQDMLAL <Va><d>, <Vb><n>, <Vb><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');
```

Vector



SQDMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
```

Assembler Symbols

- 2
- Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<b>Q</b>	<b>2</b>
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

<b>size</b>	<b>&lt;Ta&gt;</b>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

<b>size</b>	<b>Q</b>	<b>&lt;Tb&gt;</b>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in "size":

<b>size</b>	<b>&lt;Va&gt;</b>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in "size":

<b>size</b>	<b>&lt;Vb&gt;</b>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d, 2*datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SQDMLSL, SQDMLSL2 (by element)

Signed saturating Doubling Multiply-Subtract Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

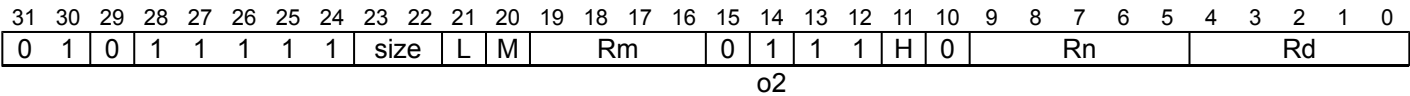
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR.QC* is set.

The *SQDMLSL* instruction extracts vector elements from the lower half of the first source register. The *SQDMLSL2* instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Scalar* and *Vector*

### Scalar



*SQDMLSL* <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]

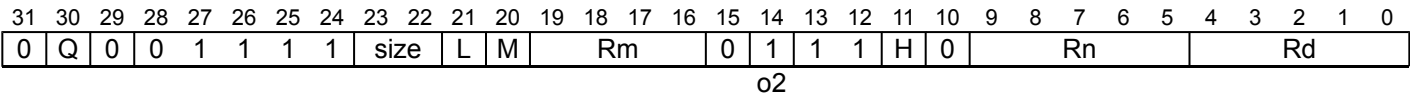
```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
```

### Vector



`SQDMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]`

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:



size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in "size:L:H:M":

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d, 2*datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMLSL, SQDMLSL2 (vector)

Signed saturating Doubling Multiply-Subtract Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

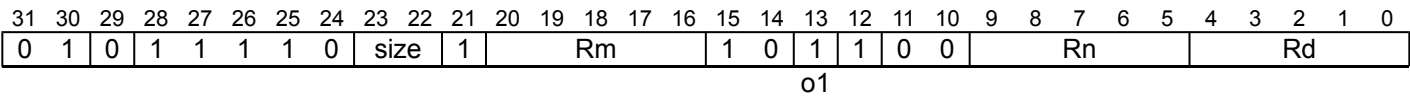
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR.QC* is set.

The *SQDMLSL* instruction extracts each source vector from the lower half of each source register. The *SQDMLSL2* instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Scalar* and *Vector*

Scalar



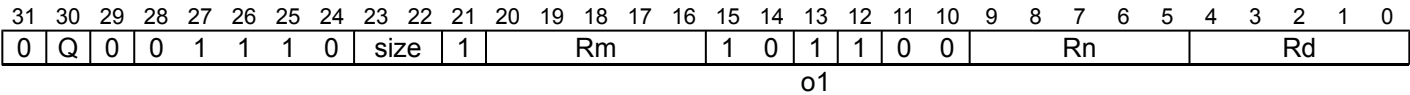
SQDMLSL <Va><d>, <Vb><n>, <Vb><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');
```

Vector



SQDMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
```

Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<b>Q</b>	<b>2</b>
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

<b>size</b>	<b>&lt;Ta&gt;</b>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

<b>size</b>	<b>Q</b>	<b>&lt;Tb&gt;</b>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in "size":

<b>size</b>	<b>&lt;Va&gt;</b>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in "size":

<b>size</b>	<b>&lt;Vb&gt;</b>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d, 2*datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SQDMULH (by element)

Signed saturating Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see [SQRDMULH](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	size	L	M				Rm		1	1	0	0	H	0				Rn				Rd		
op																															

**SQDMULH** <V><d>, <V><n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M	Rm				1	1	0	0	H	0	Rn				Rd						
op																															

**SQDMULH** <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean round = (op == '1');
```

## Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) result;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = 2 * element1 * element2;
    product = RShr(product, esize, round);
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMULH (vector)

Signed saturating Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

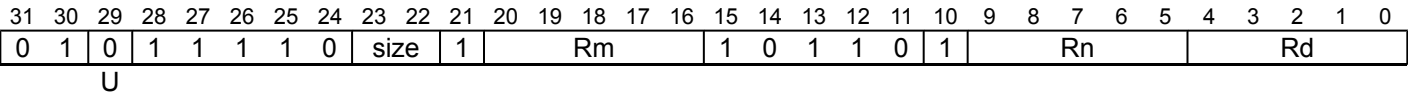
The results are truncated. For rounded results, see [SQRDMULH](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

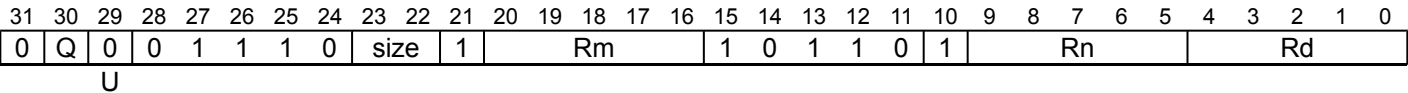
Scalar



SQDMULH <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = (U == '1');
```

Vector



SQDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.



<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = 2 * element1 * element2;
    product = RShr(product, esize, rounding);
    (Elem[result, e, esize], sat) = SignedSatQ(product, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SQDMULL, SQDMULL2 (by element)

Signed saturating Doubling Multiply Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

The [SQDMULL](#) instruction extracts the first source vector from the lower half of the first source register. The [SQDMULL2](#) instruction extracts the first source vector from the upper half of the first source register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	size	L	M					Rm		1	0	1	1	H	0								Rd	

[SQDMULL](#) <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M					Rm		1	0	1	1	H	0								Rd	

[SQDMULL{2}](#) <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

## Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- <Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

- <Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

- <Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- <Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

- <Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0 : Rm
10	M : Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

- <Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2 * esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d, 2*datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMULL, SQDMULL2 (vector)

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR.QC* is set.

The *SQDMULL* instruction extracts each source vector from the lower half of each source register. The *SQDMULL2* instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Scalar* and *Vector*

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	1				Rm			1	1	0	1	0	0				Rn				Rd		

SQDMULL <Va><d>, <Vb><n>, <Vb><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1				Rm			1	1	0	1	0	0				Rn				Rd		

SQDMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in "size":

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in "size":

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2 * esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d, 2*datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQNEG

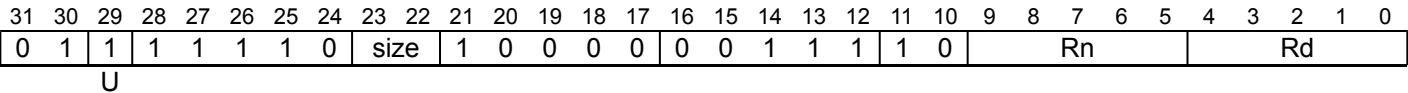
Signed saturating Negate. This instruction reads each vector element from the source SIMD&FP register, negates each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR.QC* is set.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Scalar* and *Vector*

Scalar

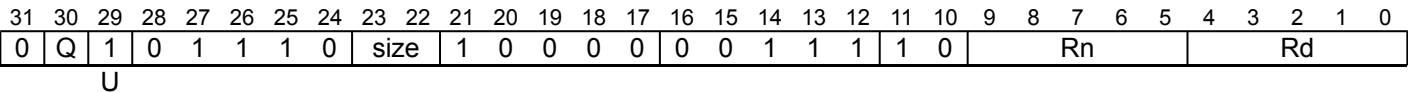


SQNEG <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector



SQNEG <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SQRDMLAH (by element)

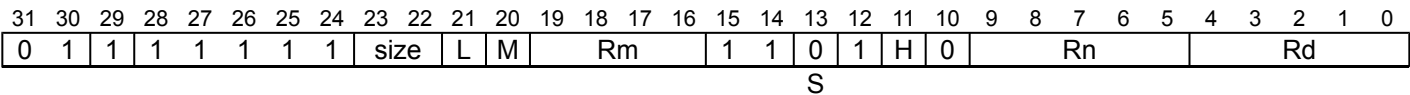
Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSR.QC](#), is set if saturation occurs.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar  
(FEAT\_RDM)



SQRDMLAH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

```
if !HaveQRDMLAHExt() then UNDEFINED;

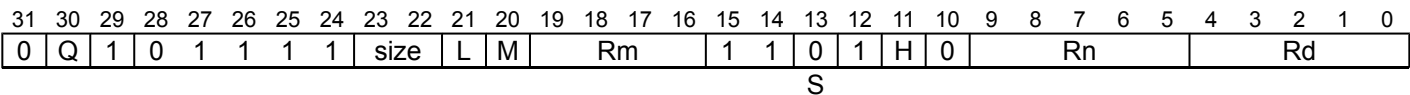
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector  
(FEAT\_RDM)



SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer element3;
integer accum;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = (element3 << esize) - 2 * (element1 * element2);
    else
        accum = (element3 << esize) + 2 * (element1 * element2);
    accum = RShr(accum, esize, rounding);
    (Elem[result, e, esize], sat) = SignedSatQ(accum, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

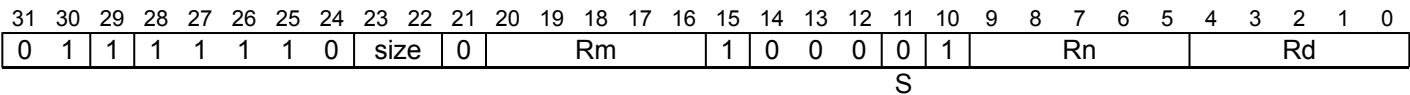
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQRDMLAH (vector)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded. If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSR.QC](#), is set if saturation occurs. Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar  
(FEAT\_RDM)

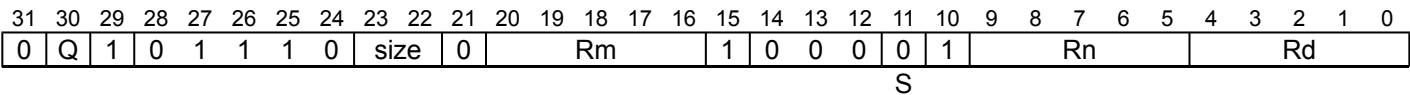


SQRDMLAH <V><d>, <V><n>, <V><m>

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector  
(FEAT\_RDM)



SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer element3;
integer accum;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = (element3 << esize) - 2 * (element1 * element2);
    else
        accum = (element3 << esize) + 2 * (element1 * element2);
    accum = RShr(accum, esize, rounding);
    (Elem[result, e, esize], sat) = SignedSatQ(accum, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

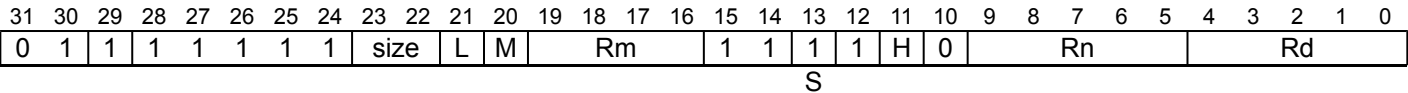
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQRDMLSH (by element)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded. If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSR.QC](#), is set if saturation occurs. Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar  
(FEAT\_RDM)



SQRDMLSH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

```
if !HaveQRDMLAHExt() then UNDEFINED;

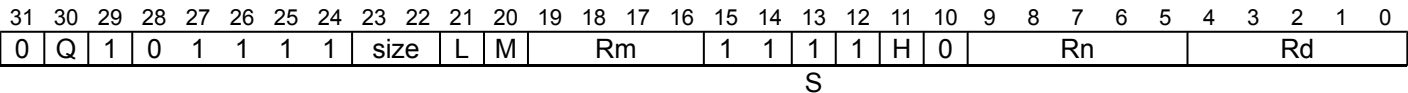
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector  
(FEAT\_RDM)



SQRDMLSH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxsize) operand2 = V[m, idxsize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer element3;
integer accum;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = (element3 << esize) - 2 * (element1 * element2);
    else
        accum = (element3 << esize) + 2 * (element1 * element2);
    accum = RShr(accum, esize, rounding);
    (Elem[result, e, esize], sat) = SignedSatQ(accum, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

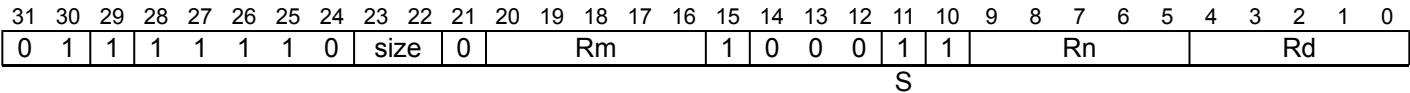
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SQRDMLSH (vector)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded. If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSR.QC](#), is set if saturation occurs. Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar  
(FEAT\_RDM)

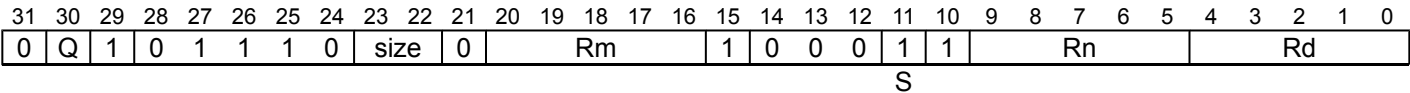


SQRDMLSH <V><d>, <V><n>, <V><m>

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector  
(FEAT\_RDM)



SQRDMLSH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer element3;
integer accum;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = (element3 << esize) - 2 * (element1 * element2);
    else
        accum = (element3 << esize) + 2 * (element1 * element2);
    accum = RShr(accum, esize, rounding);
    (Elem[result, e, esize], sat) = SignedSatQ(accum, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SQRDMULH (by element)

Signed saturating Rounding Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see [SQDMULH](#).

If any of the results overflows, they are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	size	L	M				Rm		1	1	0	1	H	0				Rn					Rd	
																	op														

**SQRDMULH** [<V><d>](#), [<V><n>](#), [<Vm>.<Ts>](#)[[<index>](#)]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	size	L	M	Rm				1	1	0	1	H	0	Rn				Rd						
op																															

**SQRDMULH** [<Vd>.<T>](#), [<Vn>.<T>](#), [<Vm>.<Ts>](#)[[<index>](#)]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean round = (op == '1');
```

## Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) result;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = 2 * element1 * element2;
    product = RShr(product, esize, round);
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQRDMULH (vector)

Signed saturating Rounding Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

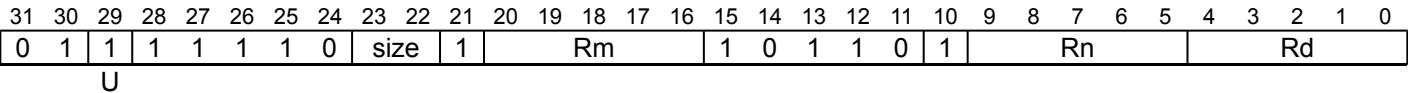
The results are rounded. For truncated results, see SQDMULH.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

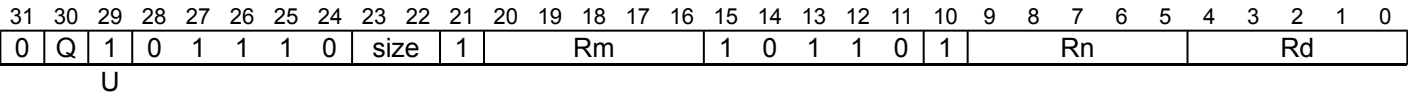
Scalar



SQRDMULH <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = (U == '1');
```

Vector



SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = 2 * element1 * element2;
    product = RShr(product, esize, rounding);
    (Elem[result, e, esize], sat) = SignedSatQ(product, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

SQRSHL

Signed saturating Rounding Shift Left (register). This instruction takes each vector element in the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

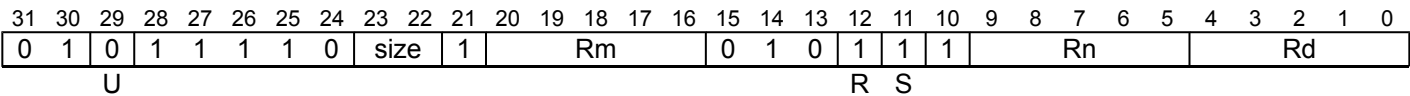
If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see [SQSHL](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

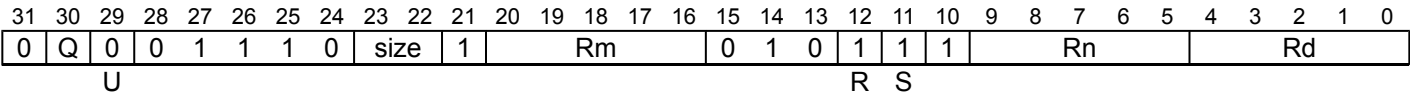
Scalar



SQRSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector



SQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:



size	<V>
00	B
01	H
10	S
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);

    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# SQRSHRN, SQRSHRN2

Signed saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see [SQSHRN](#).

The `SQRSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQRSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit `FPSR.QC` is set.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

## Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			1	0	0	1	1	1	Rn					Rd				
U									immh									op													

`SQRSHRN <Vb><d>, <Va><n>, #<shift>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

## Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000			immb			1	0	0	1	1	1	Rn					Rd					
U									immh									op													

`SQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

## Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- <Tb> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- <Ta> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

- <Vb> Is the destination width specifier, encoded in "immh":

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.

- <Va> Is the source width specifier, encoded in "immh":

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

- <shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in "immh:immb":

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = RShr(Int(Elem[operand, e, 2*esize], unsigned), shift, round);
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# SQRSHRUN, SQRSHRUN2

Signed saturating Rounded Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are rounded. For truncated results, see [SQSHRUN](#).

The SQRSHRUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR EL1](#), [CPTR EL2](#), and [CPTR EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

## Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000			immb			1	0	0	0	1	1	Rn				Rd						
immh										op																					

```
SQRSHRUN <Vb><d>, <Va><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

## Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000			immb			1	0	0	0	1	1	Rn				Rd						
immh										op																					

```
SQRSHRUN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

## Assembler Symbols

- 2
- Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in "immh":

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in "immh":

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in "immh:immb":

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in "immh:immb":

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = RShr(SInt(Elem[operand, e, 2*esize]), shift, round);
    (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SQSHL (immediate)

Signed saturating Shift Left (immediate). This instruction reads each vector element in the source SIMD&FP register, shifts each result by an immediate value, places the final result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see [UQRSHL](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			0	1	1	1	0	1	Rn				Rd					
U				immh								op																			

**SQSHL** <v><d>, <v><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			0	1	1	1	0	1	Rn				Rd					
U				immh								op																			

**SQSHL** <vd>.<T>, <vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```



Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(UInt (immh:immb)-8)
001x	(UInt (immh:immb)-16)
01xx	(UInt (immh:immb)-32)
1xxx	(UInt (immh:immb)-64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(UInt (immh:immb)-8)
001x	(UInt (immh:immb)-16)
01xx	(UInt (immh:immb)-32)
1xxx	(UInt (immh:immb)-64)

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

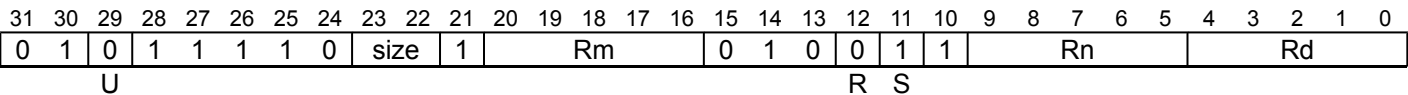
V[d, datasize] = result;
```



SQSHL (register)

Signed saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see [SQRSHL](#). If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set. Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. It has encodings from 2 classes: [Scalar](#) and [Vector](#)

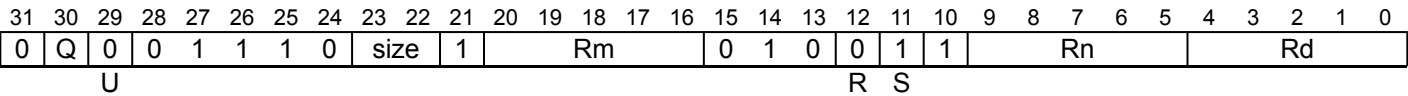
Scalar



SQSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector



SQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);

    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;

```

# SQSHLU

Signed saturating Shift Left Unsigned (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, shifts each value by an immediate value, saturates the shifted result to an unsigned integer value, places the result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see [UQRSHL](#).

If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

## Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			0	1	1	0	0	1	Rn				Rd					
U									immh				op																		

**SQSHLU** <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

## Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	1	1	0	0	1	Rn				Rd					
U									immh				op																		

**SQSHLU** <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

## Assembler Symbols

<V> Is a width specifier, encoded in "immh":

immh	<V>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in "immh:immb":

immh	<shift>
0000	RESERVED
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in "immh:immb":

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```



## SQSHRN, SQSHRN2

Signed saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts and truncates each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. For rounded results, see [SQRSHRN](#).

The `SQSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit `FPSR.QC` is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0	!= 0000				immb			1	0	0	1	0	1	Rn				Rd					
U									immh													op									

`SQSHRN <Vb><d>, <Va><n>, #<shift>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	!= 0000				immb			1	0	0	1	0	1	Rn				Rd					
U									immh													op									

`SQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```



## Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- <Tb> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- <Ta> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

- <Vb> Is the destination width specifier, encoded in "immh":

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.

- <Va> Is the source width specifier, encoded in "immh":

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

- <shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in "immh:immb":

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = RShr(Int(Elem[operand, e, 2*esize], unsigned), shift, round);
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SQSHRUN, SQSHRUN2

Signed saturating Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see [SQRSHRUN](#).

The `SQSHRUN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQSHRUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit `FPSR.QC` is set.

Depending on the settings in the [CPACR EL1](#), [CPTR EL2](#), and [CPTR EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			1	0	0	0	0	1	Rn				Rd					
immh										op																					

`SQSHRUN <Vb><d>, <Va><n>, #<shift>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	0	0	0	0	0	1	Rn				Rd				
immh										op																					

`SQSHRUN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

### Assembler Symbols

- 2
- Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in "immh":

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in "immh":

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in "immh:immb":

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in "immh:immb":

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = RShr(SInt(Elem[operand, e, 2*esize]), shift, round);
    (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQSUB

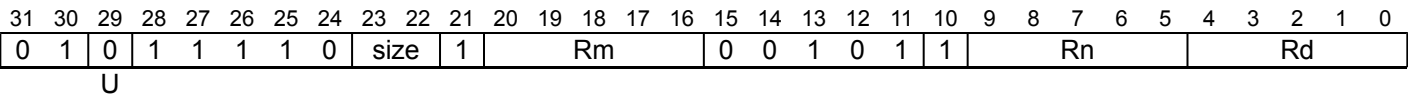
Signed saturating Subtract. This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR.QC* is set.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Scalar* and *Vector*

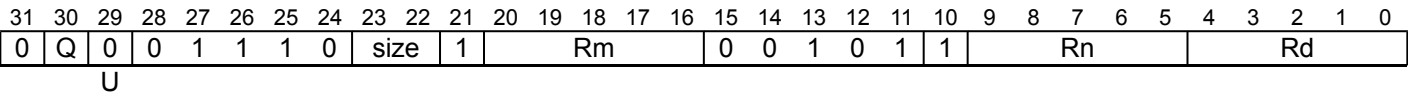
Scalar



SQSUB <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector



SQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

## SQXTN, SQXTN2

Signed saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates the value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. All the values in this instruction are signed integer values.

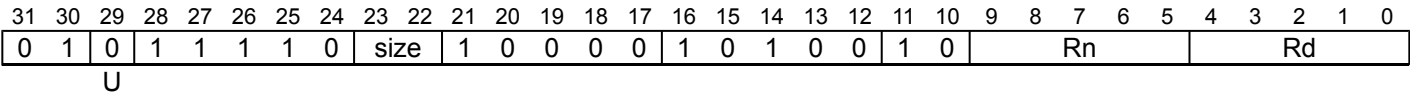
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

The [SQXTN](#) instruction writes the vector to the lower half of the destination register and clears the upper half, while the [SQXTN2](#) instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar



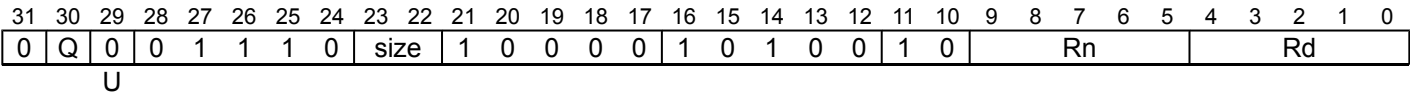
[SQXTN](#) [<Vb><d>](#), [<Va><n>](#)

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;

boolean unsigned = (U == '1');
```

### Vector



[SQXTN{2}](#) [<Vd>.<Tb>](#), [<Vn>.<Ta>](#)

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

### Assembler Symbols

- 2
- Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:



Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vb> Is the destination width specifier, encoded in "size":

size	<Vb>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in "size":

size	<Va>
00	H
01	S
10	D
11	RESERVED

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;

```

SQXTUN, SQXTUN2

Signed saturating extract Unsigned Narrow. This instruction reads each signed integer value in the vector of the source SIMD&FP register, saturates the value to an unsigned integer value that is half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

If saturation occurs, the cumulative saturation bit *FPSR.QC* is set.

The *SQXTUN* instruction writes the vector to the lower half of the destination register and clears the upper half, while the *SQXTUN2* instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Scalar* and *Vector*

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size	1	0	0	0	0	0	1	0	0	1	0	1	0	Rn				Rd					

SQXTUN <Vb><d>, <Va><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size	1	0	0	0	0	1	0	0	1	0	1	0	Rn					Rd					

SQXTUN{2} <Vd>.<Tb>, <Vn>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vb> Is the destination width specifier, encoded in "size":

size	<Vb>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in "size":

size	<Va>
00	H
01	S
10	D
11	RESERVED

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = UnsignedSatQ(SInt(element), esize);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

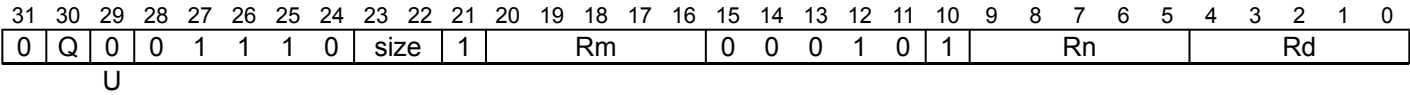
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SRHADD

Signed Rounding Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see [SHADD](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SRHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = (element1 + element2 + 1) >> 1;
    Elem[result, e, esize] = sum<esize-1:0>;

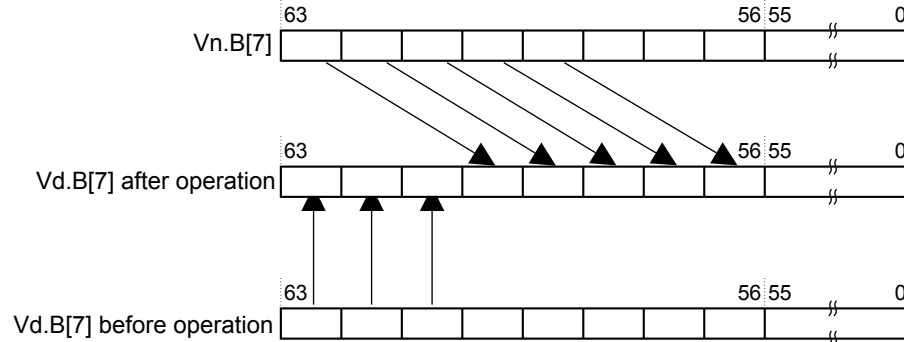
V[d, datasize] = result;
```



## SRI

Shift Right and Insert (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the right of each vector element of the source register are lost.

The following figure shows an example of the operation of shift right by 3 for an 8-bit vector element.



Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			0	1	0	0	0	1	Rn				Rd					
immh																															

SRI <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	1	0	0	0	1	Rn				Rd					
immh																															

SRI <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
```

## Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2 = V[d, datasize];
bits(datasize) result;
bits(esize) mask = LSR(Ones(esize), shift);
bits(esize) shifted;

for e = 0 to elements-1
    shifted = LSR(Elem[operand, e, esize], shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

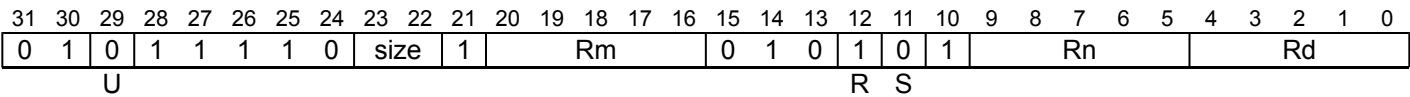
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SRSHL

Signed Rounding Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see [SSHL](#). Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. It has encodings from 2 classes: [Scalar](#) and [Vector](#)

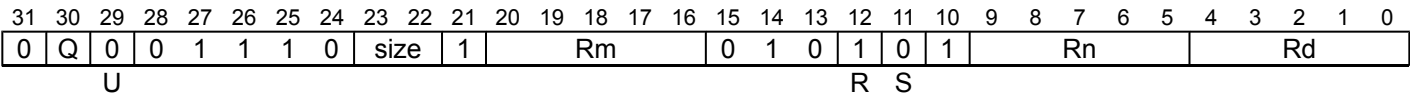
Scalar



SRSHL <v><d>, <v><n>, <v><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector



SRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);

    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;

```

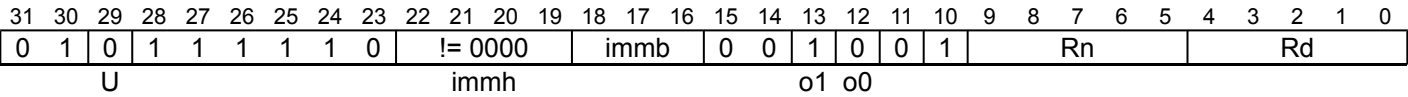
SRSHR

Signed Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see [SSHR](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar



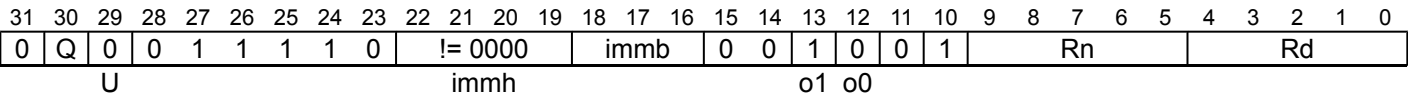
SRSHR <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector



SRSHR <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element = RShr(Int(Elem[operand, e, esize], unsigned), shift, round);
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d, datasize] = result;

```

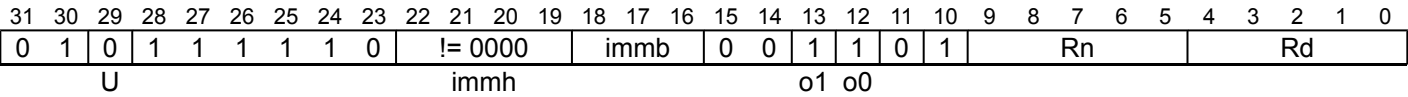
SRSRA

Signed Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see [SSRA](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar



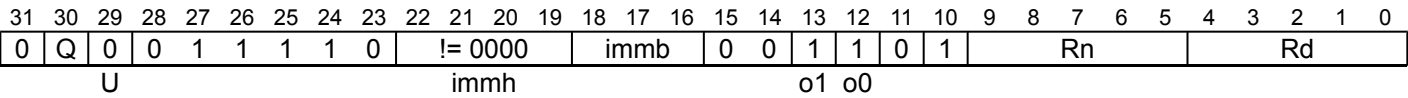
SRSRA <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector



SRSRA <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element = RShr(Int(Elem[operand, e, esize], unsigned), shift, round);
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d, datasize] = result;

```

SSHL

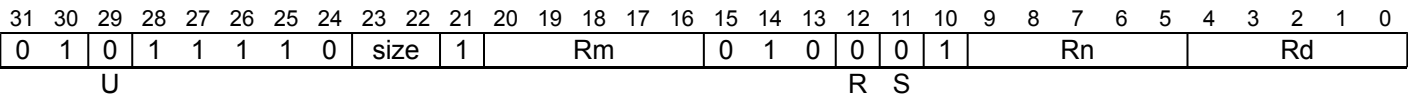
Signed Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts each value by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see [SRSHL](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

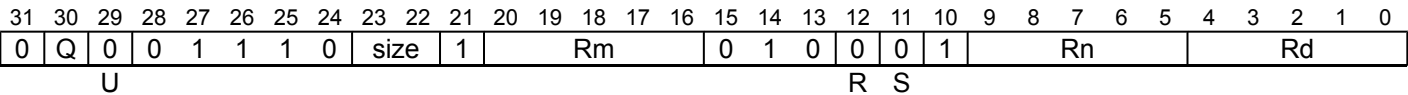
Scalar



SSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector



SSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);

    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;

```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



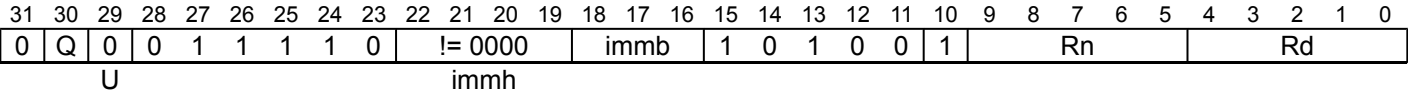
SSHLL, SSHLL2

Signed Shift Left Long (immediate). This instruction reads each vector element from the source SIMD&FP register, left shifts each vector element by the specified shift amount, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SSHLL instruction extracts vector elements from the lower half of the source register. The SSHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR EL1, CPTR EL2, and CPTR EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias SXTL, SXTL2.



```
SSHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimh);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<shift> Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(UInt (immh:immb)-8)
001x	(UInt (immh:immb)-16)
01xx	(UInt (immh:immb)-32)
1xxx	RESERVED

## Alias Conditions

Alias	Is preferred when
<a href="#">SXTL, SXTL2</a>	immb == '000' && <a href="#">BitCount</a> (immh) == 1

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part, datasize];
bits(datasize*2) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d, datasize*2] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

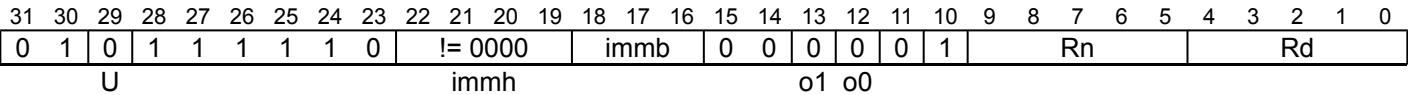
SSHR

Signed Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see [SRSHR](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar



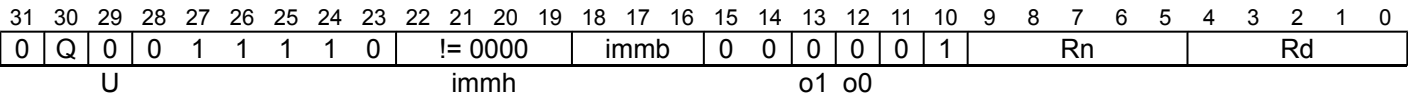
SSHR <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector



SSHR <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt (immh:immb) )

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt (immh:immb) )
001x	(32-UInt (immh:immb) )
01xx	(64-UInt (immh:immb) )
1xxx	(128-UInt (immh:immb) )

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element = RShr(Int(Elem[operand, e, esize], unsigned), shift, round);
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d, datasize] = result;

```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

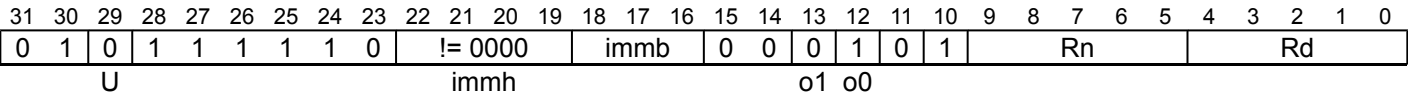
SSRA

Signed Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see [SRSRA](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar



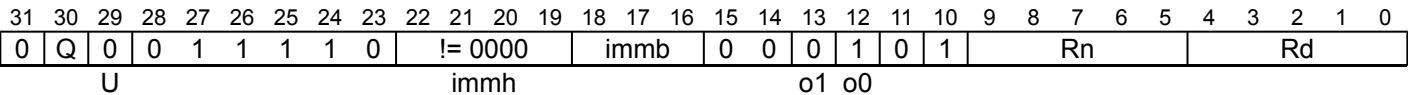
SSRA <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector



SSRA <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt (immh:immb) )

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt (immh:immb) )
001x	(32-UInt (immh:immb) )
01xx	(64-UInt (immh:immb) )
1xxx	(128-UInt (immh:immb) )

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element = RShr(Int(Elem[operand, e, esize], unsigned), shift, round);
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d, datasize] = result;

```

### Operational information

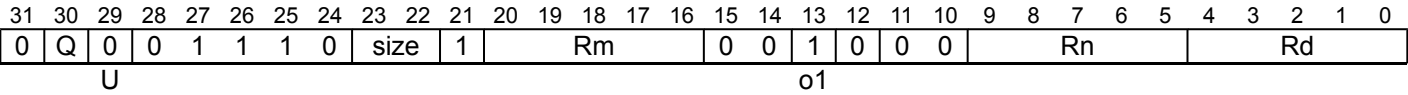
- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

SSUBL, SSUBL2

Signed Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are twice as long as the source vector elements.

The SSUBL instruction extracts each source vector from the lower half of each source register. The SSUBL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



```
SSUBL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

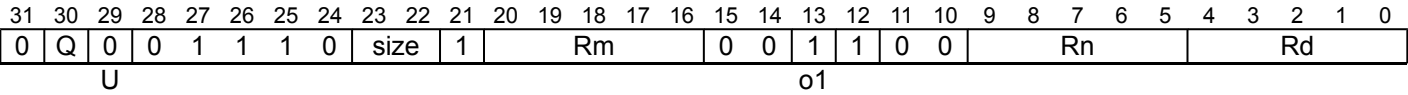


SSUBW, SSUBW2

Signed Subtract Wide. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The SSUBW instruction extracts the second source vector from the lower half of the second source register. The SSUBW2 instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



```
SSUBW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST1 (multiple structures)

Store multiple single-element structures from one, two, three, or four registers. This instruction stores elements to memory from one, two, three, or four SIMD&FP registers, without interleaving. Every element of each register is stored.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	x	x	1	x	size	Rn				Rt						
L										opcode																					

One register (opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>]
```

Two registers (opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Three registers (opcode == 0110)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

Four registers (opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	0	0	Rm				x	x	1	x	size	Rn				Rt							
L										opcode																					

One register, immediate offset (Rm == 11111 && opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

One register, register offset (Rm != 11111 && opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

Two registers, immediate offset (Rm == 11111 && opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Two registers, register offset (Rm != 11111 && opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Three registers, immediate offset (Rm == 11111 && opcode == 0110)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Three registers, register offset (Rm != 11111 && opcode == 0110)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Four registers, immediate offset (Rm == 11111 && opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Four registers, register offset (Rm != 11111 && opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#8
1	#16

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#16
1	#32

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#24
1	#48

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

### Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp\_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, accdesc];
                V[tt, datasize] = rval;
            else // memop == MemOp\_STORE
                Mem[address+offs, ebytes, accdesc] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST1 (single structure)

Store a single-element structure from one lane of one register. This instruction stores the specified element of a SIMD&FP register to memory.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	0	0	0	0	0	0	x	x	0	S	size	Rn					Rt					
L R										o2 opcode																					

### 8-bit (opcode == 000)

ST1 { <Vt>.B }[<index>], [<Xn|SP>]

### 16-bit (opcode == 010 && size == x0)

ST1 { <Vt>.H }[<index>], [<Xn|SP>]

### 32-bit (opcode == 100 && size == 00)

ST1 { <Vt>.S }[<index>], [<Xn|SP>]

### 64-bit (opcode == 100 && S == 0 && size == 01)

ST1 { <Vt>.D }[<index>], [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	0	0	Rm					x	x	0	S	size	Rn					Rt					
L R										opcode																					

#### 8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>], #1
```

#### 8-bit, register offset (Rm != 11111 && opcode == 000)

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>
```

#### 16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>], #2
```

#### 16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>
```

#### 32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>], #4
```

#### 32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>
```

#### 64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>], #8
```

#### 64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

### Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.



## Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);    // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp\_LOAD else MemOp\_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp\_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, accdesc];
            V[t, 128] = rval;
        else // memop == MemOp\_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, accdesc] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

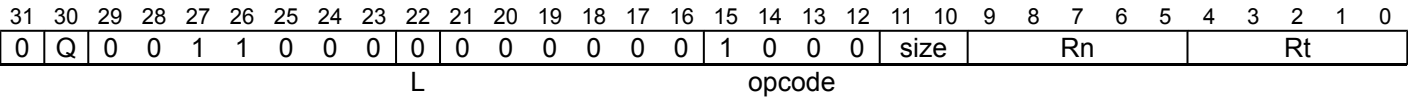
ST2 (multiple structures)

Store multiple 2-element structures from two registers. This instruction stores multiple 2-element structures from two SIMD&FP registers to memory, with interleaving. Every element of each register is stored.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

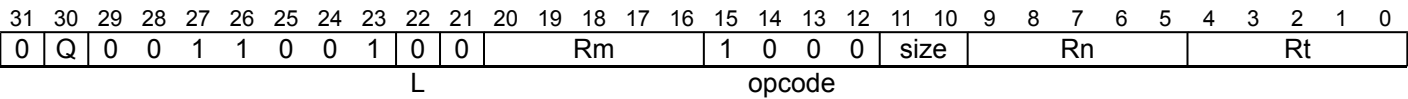
No offset



```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

- <Xn|SP>

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm>

Is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#16
1	#32
- <Xm>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;

```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp\_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, accdesc];
                V[tt, datasize] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, accdesc] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST2 (single structure)

Store single 2-element structure from one lane of two registers. This instruction stores a 2-element structure to memory from corresponding elements of two SIMD&FP registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	1	0	0	0	0	0	x	x	0	S	size	Rn				Rt						
L										R	o2										opcode										

### 8-bit (opcode == 000)

ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]

### 16-bit (opcode == 010 && size == x0)

ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]

### 32-bit (opcode == 100 && size == 00)

ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]

### 64-bit (opcode == 100 && S == 0 && size == 01)

ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	0	1	Rm				x	x	0	S	size	Rn				Rt							
L										R	opcode																				

### 8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2
```

### 8-bit, register offset (Rm != 11111 && opcode == 000)

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4
```

### 16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8
```

### 32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16
```

### 64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);  // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp\_LOAD else MemOp\_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```



## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp\_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, accdesc];
            V[t, 128] = rval;
        else // memop == MemOp\_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, accdesc] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

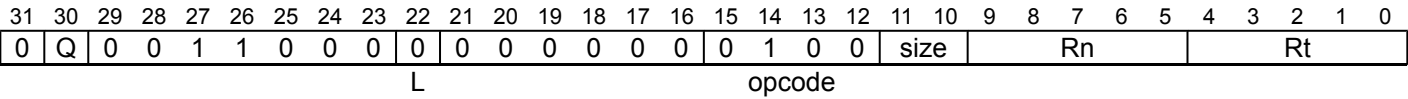
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST3 (multiple structures)

Store multiple 3-element structures from three registers. This instruction stores multiple 3-element structures to memory from three SIMD&FP registers, with interleaving. Every element of each register is stored. Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. It has encodings from 2 classes: [No offset](#) and [Post-index](#)

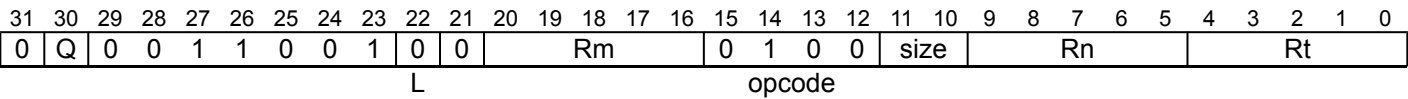
No offset



```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

- <Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the post-index immediate offset, encoded in "Q":
 

Q	<imm>
0	#24
1	#48
- <Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;      // number of iterations
integer selem;    // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;      // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;      // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;      // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;      // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;      // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;      // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;      // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;

```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp\_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, accdesc];
                V[tt, datasize] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, accdesc] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST3 (single structure)

Store single 3-element structure from one lane of three registers. This instruction stores a 3-element structure to memory from corresponding elements of three SIMD&FP registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	0	0	0	0	0	0	x	x	1	S	size	Rn					Rt					
L R										o2 opcode																					

8-bit (opcode == 001)

ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>]

16-bit (opcode == 011 && size == x0)

ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]

32-bit (opcode == 101 && size == 00)

ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]

64-bit (opcode == 101 && S == 0 && size == 01)

ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	0	0	Rm					x	x	1	S	size	Rn					Rt					
L R										opcode																					

### 8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3
```

### 8-bit, register offset (Rm != 11111 && opcode == 001)

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6
```

### 16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12
```

### 32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24
```

### 64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
      scale = 3;
  end

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp\_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, accdesc];
            V[t, 128] = rval;
        else // memop == MemOp\_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, accdesc] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



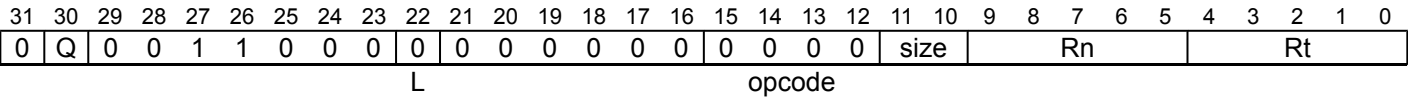
ST4 (multiple structures)

Store multiple 4-element structures from four registers. This instruction stores multiple 4-element structures to memory from four SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

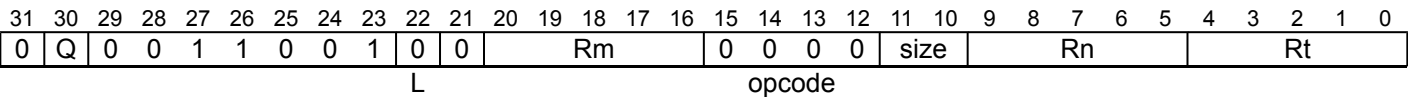
No offset



ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

- <Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- <Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the post-index immediate offset, encoded in "Q":
 

Q	<imm>
0	#32
1	#64
- <Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;

```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp\_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, accdesc];
                V[tt, datasize] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, accdesc] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST4 (single structure)

Store single 4-element structure from one lane of four registers. This instruction stores a 4-element structure to memory from corresponding elements of four SIMD&FP registers.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [No offset](#) and [Post-index](#)

### No offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	0	1	0	0	0	0	0	x	x	1	S	size	Rn				Rt						
L R										o2 opcode																					

### 8-bit (opcode == 001)

ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]

### 16-bit (opcode == 011 && size == x0)

ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]

### 32-bit (opcode == 101 && size == 00)

ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]

### 64-bit (opcode == 101 && S == 0 && size == 01)

ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	0	1	Rm				x	x	1	S	size	Rn				Rt							
L R										opcode																					

### 8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

### 8-bit, register offset (Rm != 11111 && opcode == 001)

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```

### 16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

### 16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

### 32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

### 32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

### 64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

### 64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

## Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4>	Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

## Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, nontemporal, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp\_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, accdesc];
            V[t, 128] = rval;
        else // memop == MemOp\_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, accdesc] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

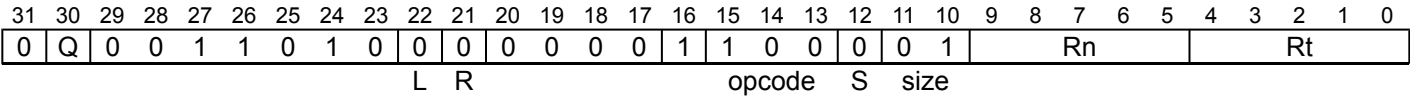
## STL1 (SIMD&FP)

Store-Release a single-element structure from one lane of one register. This instruction stores the specified element of a SIMD&FP register to memory.

The instruction also has memory ordering semantics, as described in [Load-Acquire](#), [Load-AcquirePC](#), and [Store-Release](#). For information about memory accesses, see [Load/Store addressing modes](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### 64-bit (FEAT\_LRCPC3)



STL1 { <Vt>.D } [<index>], [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = wback || n != 31;
```

### Assembler Symbols

- <Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- <index> Is the element index, encoded in "Q".
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q); // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```



## Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMDAcqRel(memop, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp\_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, accdesc];
            V[t, 128] = rval;
        else // memop == MemOp\_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, accdesc] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLUR (SIMD&FP)

Store-Release SIMD&FP Register (unscaled offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an optional immediate offset. The instruction has memory ordering semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*. Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Unscaled offset  
(FEAT\_LRCPC3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	0	1	1	1	0	1	x	0	0	imm9										1	0	Rn				Rt					
opc																															

8-bit (size == 00 && opc == 00)

```
STLUR <Bt>, [<Xn|SP>{, #<sim>}]
```

16-bit (size == 01 && opc == 00)

```
STLUR <Ht>, [<Xn|SP>{, #<sim>}]
```

32-bit (size == 10 && opc == 00)

```
STLUR <St>, [<Xn|SP>{, #<sim>}]
```

64-bit (size == 11 && opc == 00)

```
STLUR <Dt>, [<Xn|SP>{, #<sim>}]
```

128-bit (size == 00 && opc == 10)

```
STLUR <Qt>, [<Xn|SP>{, #<sim>}]
```

```
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp\_LOAD else MemOp\_STORE;
integer datasize = 8 << scale;
boolean tagchecked = memop != MemOp\_PREFETCH && (n != 31);
```

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescASIMDAcqRel(memop, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp\_STORE
        data = V[t, datasize];
        Mem[address, datasize DIV 8, accdesc] = data;

    when MemOp\_LOAD
        data = Mem[address, datasize DIV 8, accdesc];
        V[t, datasize] = data;
```

## Operational information

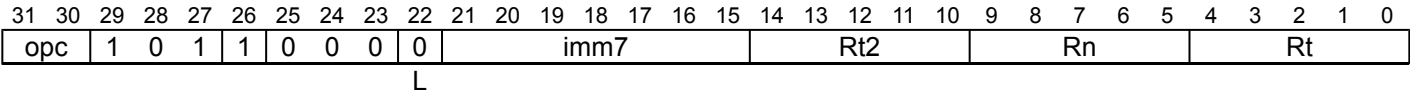
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNP (SIMD&FP)

Store Pair of SIMD&FP registers, with Non-temporal hint. This instruction stores a pair of SIMD&FP registers to memory, issuing a hint to the memory system that the access is non-temporal. The address used for the store is calculated from an address from a base register value and an immediate offset. For information about non-temporal pair instructions, see [Load/Store SIMD and Floating-point Non-temporal pair](#). Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



### 32-bit (opc == 00)

```
STNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

### 64-bit (opc == 01)

```
STNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

### 128-bit (opc == 10)

```
STNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
// Empty.
```

## Assembler Symbols

- <Dt1> Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Dt2> Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- <Qt1> Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Qt2> Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- <St1> Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- <St2> Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.  
For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.  
For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tagchecked = n != 31;
```

## Operation

```
CheckFPEnabled64();  
bits(64) address;  
bits(datasize) data1;  
bits(datasize) data2;  
constant integer dbytes = datasize DIV 8;  
  
AccessDescriptor accdesc = CreateAccDescASIMD(MemOp\_STORE, TRUE, tagchecked);  
  
if n == 31 then  
    CheckSPAlignment();  
    address = SP[];  
else  
    address = X[n, 64];  
  
address = address + offset;  
  
data1 = V[t, datasize];  
data2 = V[t2, datasize];  
Mem[address, dbytes, accdesc] = data1;  
Mem[address+dbytes, dbytes, accdesc] = data2;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

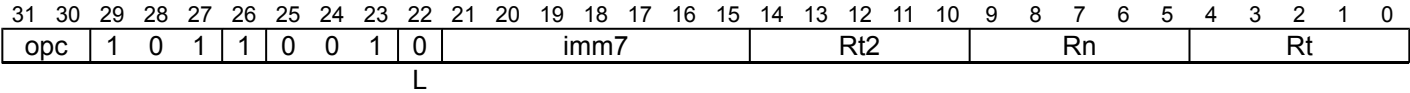
STP (SIMD&FP)

Store Pair of SIMD&FP registers. This instruction stores a pair of SIMD&FP registers to memory. The address used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Signed offset](#)

Post-index



32-bit (opc == 00)

```
STP <St1>, <St2>, [<Xn|SP>], #<imm>
```

64-bit (opc == 01)

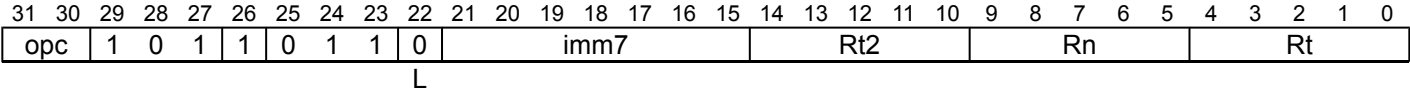
```
STP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>
```

128-bit (opc == 10)

```
STP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index



32-bit (opc == 00)

```
STP <St1>, <St2>, [<Xn|SP>, #<imm>]!
```

64-bit (opc == 01)

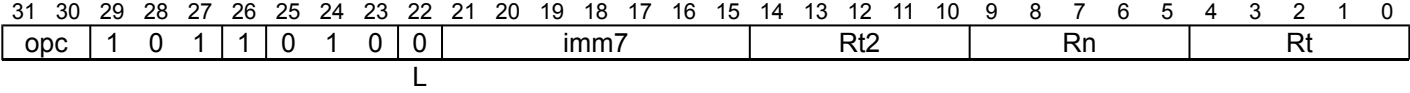
```
STP <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]!
```

128-bit (opc == 10)

```
STP <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset



### 32-bit (opc == 00)

```
STP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

### 64-bit (opc == 01)

```
STP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

### 128-bit (opc == 10)

```
STP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;  
boolean postindex = FALSE;
```

## Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.  For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.  For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.  For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.  For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16.  For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

## Shared Decode

```
integer n = UInt(Rn);  
integer t = UInt(Rt);  
integer t2 = UInt(Rt2);  
if opc == '11' then UNDEFINED;  
integer scale = 2 + UInt(opc);  
integer datasize = 8 << scale;  
bits(64) offset = LSL(SignExtend(imm7, 64), scale);  
boolean tagchecked = wback || n != 31;
```

## Operation

```
CheckFPEEnabled64();
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc = CreateAccDescASIMD(MemOp\_STORE, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

data1 = V[t, datasize];
data2 = V[t2, datasize];
Mem[address, dbytes, accdesc] = data1;
Mem[address+dbytes, dbytes, accdesc] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## STR (immediate, SIMD&FP)

Store SIMD&FP register (immediate offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an immediate offset. Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. It has encodings from 3 classes: [Post-index](#) , [Pre-index](#) and [Unsigned offset](#)

### Post-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	1	0	0	x	0	0	imm9										0	1	Rn				Rt				
opc																															

#### 8-bit (size == 00 && opc == 00)

STR <Bt>, [<Xn|SP>], #<sim>

#### 16-bit (size == 01 && opc == 00)

STR <Ht>, [<Xn|SP>], #<sim>

#### 32-bit (size == 10 && opc == 00)

STR <St>, [<Xn|SP>], #<sim>

#### 64-bit (size == 11 && opc == 00)

STR <Dt>, [<Xn|SP>], #<sim>

#### 128-bit (size == 00 && opc == 10)

STR <Qt>, [<Xn|SP>], #<sim>

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	1	0	0	x	0	0	imm9										1	1	Rn				Rt				
opc																															

8-bit (size == 00 && opc == 00)

```
STR <Bt>, [<Xn|SP>, #<simm>]!
```

16-bit (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>, #<simm>]!
```

32-bit (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>, #<simm>]!
```

64-bit (size == 11 && opc == 00)

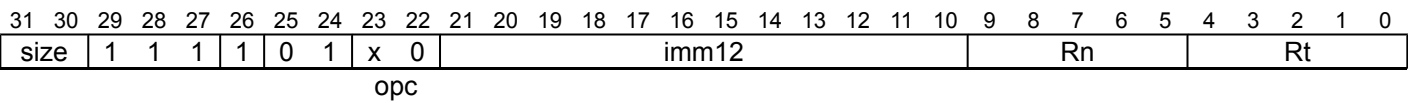
```
STR <Dt>, [<Xn|SP>, #<simm>]!
```

128-bit (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



8-bit (size == 00 && opc == 00)

```
STR <Bt>, [<Xn|SP>{, #<pimm>}]
```

16-bit (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>{, #<pimm>}]
```

32-bit (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>{, #<pimm>}]
```

64-bit (size == 11 && opc == 00)

```
STR <Dt>, [<Xn|SP>{, #<pimm>}]
```

128-bit (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

## Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<sim>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.  For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.  For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.  For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.  For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as <pimm>/16.

## Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tagchecked = memop != MemOp_PREFETCH && (wback || n != 31);
```

## Operation

```
CheckFPEnabled64();
bits(64) address;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t, datasize];
        Mem[address, datasize DIV 8, accdesc] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, accdesc];
        V[t, datasize] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STR (register, SIMD&FP)

Store SIMD&FP register (register offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	1	1	1	1	0	0	x	0	1	Rm						option			S	1	0	Rn						Rt			
opc																															

8-bit (size == 00 && opc == 00 && option != 011)

```
STR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

8-bit (size == 00 && opc == 00 && option == 011)

```
STR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

16-bit (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

32-bit (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit (size == 11 && opc == 00)

```
STR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

128-bit (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

- <Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- <Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> For the 8-bit variant: is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	SCTX

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SCTX

<amount> For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#4

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tagchecked = memop != MemOp_PREFETCH;
```

## Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
CheckFPEEnabled64();
bits(64) address;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescASIMD(memop, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp\_STORE
        data = V[t, datasize];
        Mem[address, datasize DIV 8, accdesc] = data;

    when MemOp\_LOAD
        data = Mem[address, datasize DIV 8, accdesc];
        V[t, datasize] = data;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STUR (SIMD&FP)

Store SIMD&FP register (unscaled offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an optional immediate offset. Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	1	1	1	1	0	0	x	0	0	imm9										0	0	Rn					Rt				
opc																															

8-bit (size == 00 && opc == 00)

```
STUR <Bt>, [<Xn|SP>{, #<sim>}]
```

16-bit (size == 01 && opc == 00)

```
STUR <Ht>, [<Xn|SP>{, #<sim>}]
```

32-bit (size == 10 && opc == 00)

```
STUR <St>, [<Xn|SP>{, #<sim>}]
```

64-bit (size == 11 && opc == 00)

```
STUR <Dt>, [<Xn|SP>{, #<sim>}]
```

128-bit (size == 00 && opc == 10)

```
STUR <Qt>, [<Xn|SP>{, #<sim>}]
```

```
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- <Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <sim> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tagchecked = memop != MemOp_PREFETCH && (n != 31);
```



## Operation

```
CheckFPEEnabled64();  
bits(64) address;  
bits(datasize) data;  
  
AccessDescriptor accdesc = CreateAccDescASIMD(memop, FALSE, tagchecked);  
  
if n == 31 then  
    CheckSPAlignment();  
    address = SP[];  
else  
    address = X[n, 64];  
  
address = address + offset;  
  
case memop of  
    when MemOp\_STORE  
        data = V[t, datasize];  
        Mem[address, datasize DIV 8, accdesc] = data;  
  
    when MemOp\_LOAD  
        data = Mem[address, datasize DIV 8, accdesc];  
        V[t, datasize] = data;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

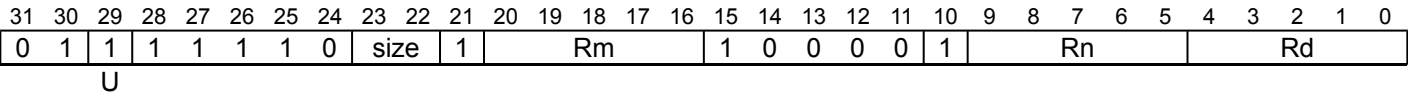
SUB (vector)

Subtract (vector). This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

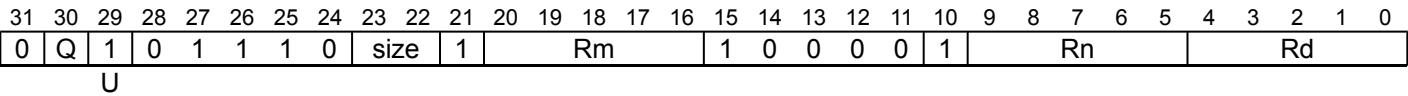
Scalar



SUB <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');
```

Vector



SUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;

V[d, datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

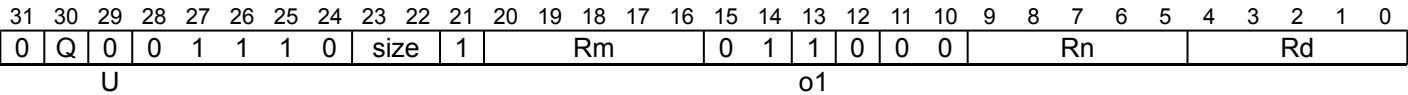
SUBHN, SUBHN2

Subtract returning High Narrow. This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values.

The results are truncated. For rounded results, see *RSUBHN*.

The SUBHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SUBHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SUBHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(2*datasize) operand2 = V[m, 2*datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, 2*esize]);
    element2 = UInt(Elem[operand2, e, 2*esize]);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = RShr(sum, esize, round);
    Elem[result, e, esize] = sum<esize-1:0>;

Vpart[d, part, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

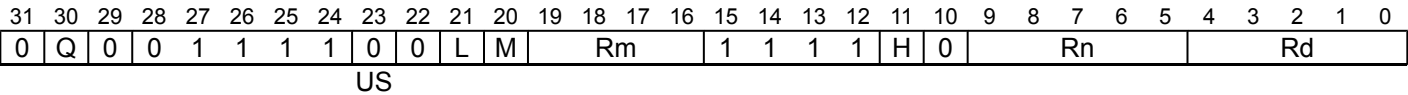
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUDOT (by element)

Dot product index form with signed and unsigned integers. This instruction performs the dot product of the four signed 8-bit integer values in each 32-bit element of the first source register with the four unsigned 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination vector.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. [ID\\_AA64ISAR1\\_EL1](#).I8MM indicates whether this instruction is supported.

Vector  
(FEAT\_I8MM)



SUDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]

```
if !HaveInt8MatMulExt() then UNDEFINED;
boolean op1_unsigned = (US == '1');
boolean op2_unsigned = (US == '0');
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

Assembler Symbols

- <Vd>

Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
- <Ta>

Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S
- <Vn>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Tb>

Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B
- <Vm>

Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
- <index>

Is the immediate index of a 32-bit group of four 8-bit values in the range 0 to 3, encoded in the "H:L" fields.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(128) operand2 = V[m, 128];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;

for e = 0 to elements-1
    bits(32) res = Elem[operand3, e, 32];
    for b = 0 to 3
        integer element1 = Int(Elem[operand1, 4*e+b, 8], op1_unsigned);
        integer element2 = Int(Elem[operand2, 4*i+b, 8], op2_unsigned);
        res = res + element1 * element2;
    Elem[result, e, 32] = res;
V[d, datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

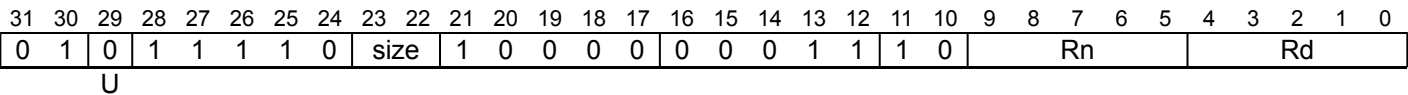
SUQADD

Signed saturating Accumulate of Unsigned value. This instruction adds the unsigned integer values of the vector elements in the source SIMD&FP register to corresponding signed integer values of the vector elements in the destination SIMD&FP register, and writes the resulting signed integer values to the destination SIMD&FP register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit `FPSR.QC` is set.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar



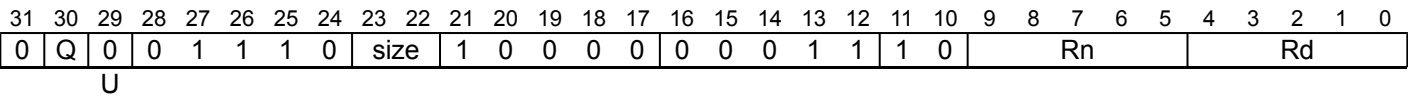
SUQADD <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean unsigned = (U == '1');
```

Vector



SUQADD <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

- <d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- <n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.



<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

bits(datasize) operand2 = V[d, datasize];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d, datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SXTL, SXTL2

Signed extend Long. This instruction duplicates each vector element in the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values. The SXTL instruction extracts the source vector from the lower half of the source register. The SXTL2 instruction extracts the source vector from the upper half of the source register. Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of SSHLL, SSHLL2. This means:

- The encodings in this description are named to match the encodings of SSHLL, SSHLL2.
- The description of SSHLL, SSHLL2 gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	Q	0	0	1	1	1	1	0	!= 0000				0	0	0	1	0	1	0	0	1	Rn				Rd						
U									immh								immb															

SXTL{2} <Vd>.<Ta>, <Vn>.<Tb>

is equivalent to

SSHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #0

and is the preferred disassembly when BitCount(immh) == 1.

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

## Operation

The description of [SSHLL](#), [SSHLL2](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

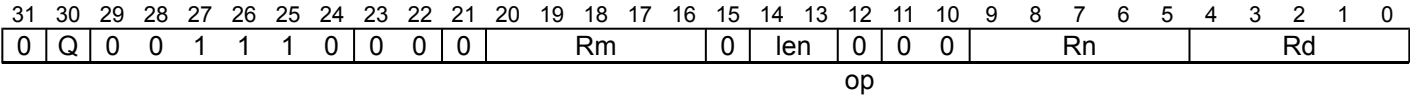
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TBL

Table vector Lookup. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the result for that lookup is 0. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Two register table (len == 01)

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

Three register table (len == 10)

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

Four register table (len == 11)

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

Single register table (len == 00)

```
TBL <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Ta> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	8B
1	16B
- <Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.  
For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.
- <Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.
- <Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.
- <Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.
- <Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m, datasize];
bits(128*regs) table = Zeros(128 * regs);
bits(datasize) result;
integer index;

// Create table from registers
for i = 0 to regs-1
    table<128*i+127:128*i> = V[n, 128];
    n = (n + 1) MOD 32;

result = if is_tbl then Zeros(datasize) else V[d, datasize];
for i = 0 to elements-1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

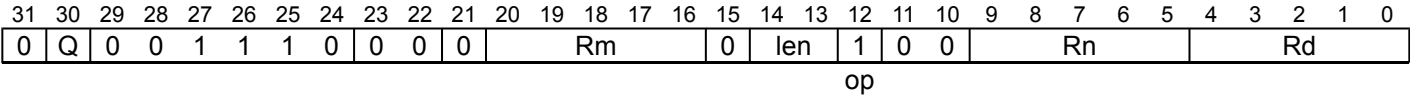
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TBX

Table vector lookup extension. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the existing value in the vector element of the destination register is left unchanged. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Two register table (len == 01)

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

Three register table (len == 10)

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

Four register table (len == 11)

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

Single register table (len == 00)

```
TBX <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Ta> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	8B
1	16B
- <Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.  
For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.
- <Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.
- <Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.
- <Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.
- <Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m, datasize];
bits(128*regs) table = Zeros(128 * regs);
bits(datasize) result;
integer index;

// Create table from registers
for i = 0 to regs-1
    table<128*i+127:128*i> = V[n, 128];
    n = (n + 1) MOD 32;

result = if is_tbl then Zeros(datasize) else V[d, datasize];
for i = 0 to elements-1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

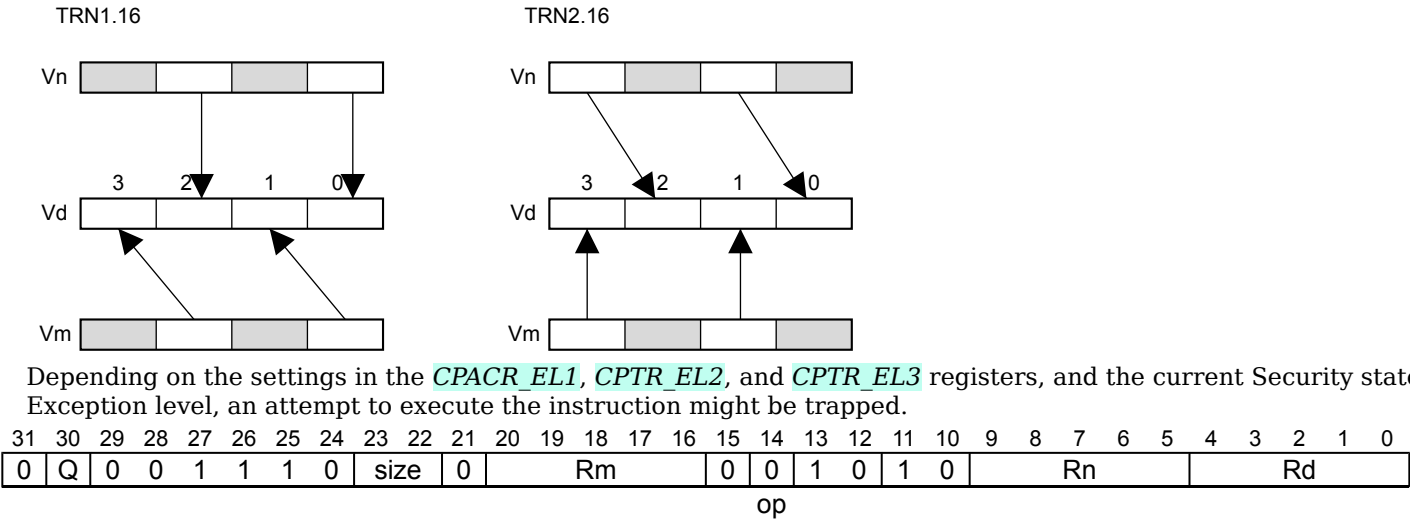
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TRN1

Transpose vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

Note

By using this instruction with TRN2, a 2 x 2 matrix can be transposed. The following figure shows an example of the operation of TRN1 and TRN2 halfword operations where Q = 0.



Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

TRN1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.



## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

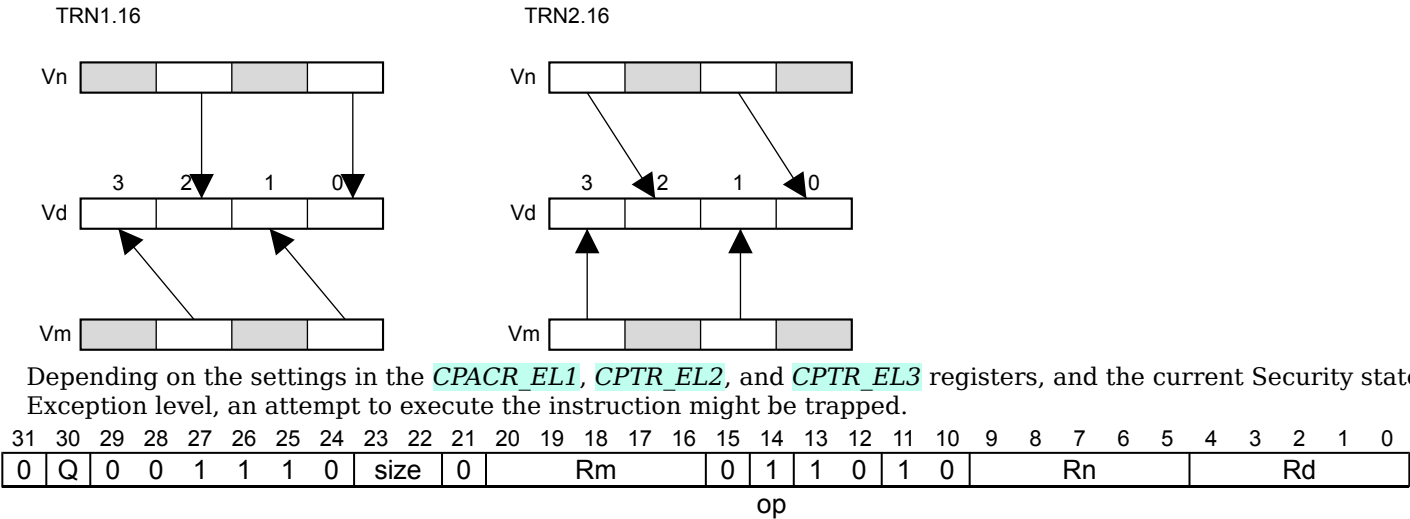
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TRN2

Transpose vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

Note

By using this instruction with TRN1, a 2 x 2 matrix can be transposed. The following figure shows an example of the operation of TRN1 and TRN2 halfword operations where Q = 0.



Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	0					Rm		0	1	1	0	1	0					Rn				Rd	
op																															

TRN2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

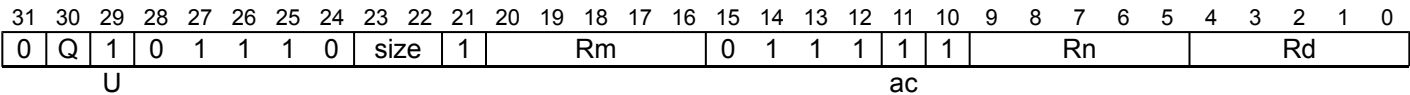
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UABA

Unsigned Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

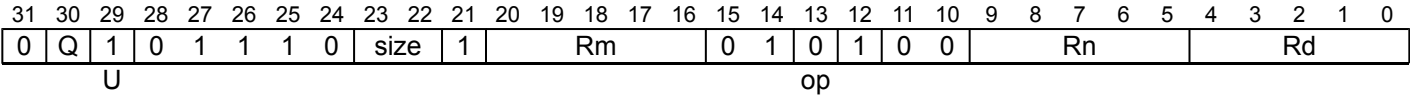
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UABAL, UABAL2

Unsigned Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UABAL instruction extracts each source vector from the lower half of each source register. The UABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR EL1, CPTR EL2, and CPTR EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UABAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d, 2*datasize] else Zeros(2 * datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d, 2*datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

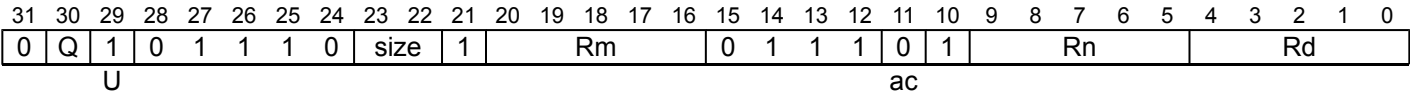
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UABD

Unsigned Absolute Difference (vector). This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d, datasize] = result;
```



## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

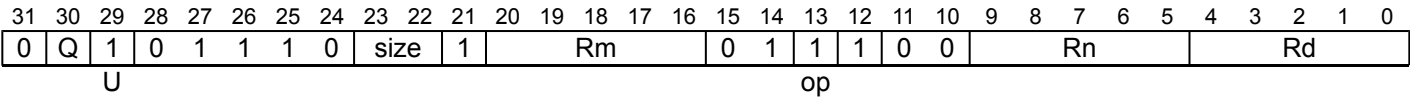
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UABDL, UABDL2

Unsigned Absolute Difference Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UABDL instruction extracts each source vector from the lower half of each source register. The UABDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR EL1, CPTR EL2, and CPTR EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d, 2*datasize] else Zeros(2 * datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d, 2*datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

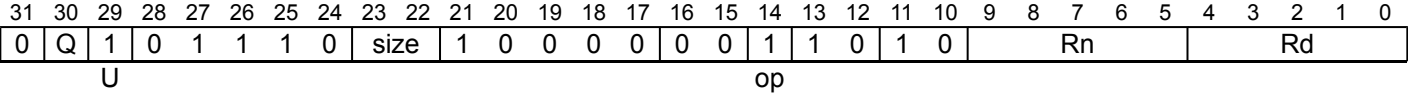
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UADALP

Unsigned Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UADALP <Vd>.<Ta>, <Vn>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d, datasize];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

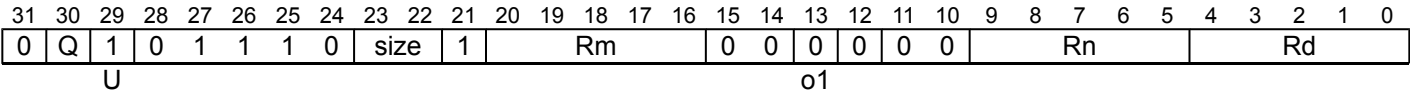
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UADDL, UADDL2

Unsigned Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UADDL instruction extracts each source vector from the lower half of each source register. The UADDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UADDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

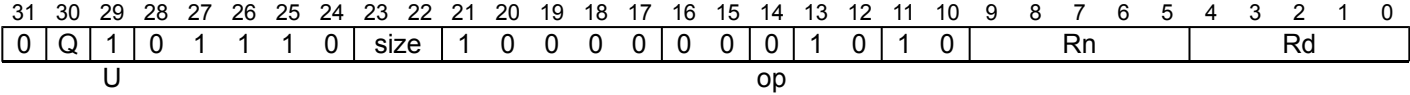
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UADDLP

Unsigned Add Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UADDLP <Vd>.<Ta>, <Vn>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED



## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d, datasize];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

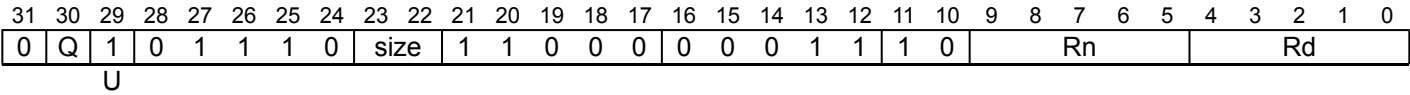
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UADDLV

Unsigned sum Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UADDLV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	H
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
integer sum;

sum = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    sum = sum + Int(Elem[operand, e, esize], unsigned);

V[d, 2*esize] = sum<2*esize-1:0>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

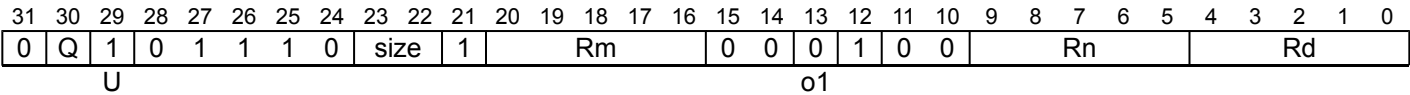
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UADDW, UADDW2

Unsigned Add Wide. This instruction adds the vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register. All the values in this instruction are unsigned integer values.

The UADDW instruction extracts vector elements from the lower half of the second source register. The UADDW2 instruction extracts vector elements from the upper half of the second source register.

Depending on the settings in the CPACR EL1, CPTR EL2, and CPTR EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



```
UADDW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UCVTF (scalar, fixed-point)

Unsigned fixed-point Convert to Floating-point (scalar). This instruction converts the unsigned value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		0	0	1	1	1	1	0	ftype		0	0	0	0	1	1	scale					Rn					Rd				
rmode										opcode																					

### 32-bit to half-precision (sf == 0 && ftype == 11) (FEAT\_FP16)

```
UCVTF <Hd>, <Wn>, #<fbits>
```

### 32-bit to single-precision (sf == 0 && ftype == 00)

```
UCVTF <Sd>, <Wn>, #<fbits>
```

### 32-bit to double-precision (sf == 0 && ftype == 01)

```
UCVTF <Dd>, <Wn>, #<fbits>
```

### 64-bit to half-precision (sf == 1 && ftype == 11) (FEAT\_FP16)

```
UCVTF <Hd>, <Xn>, #<fbits>
```

### 64-bit to single-precision (sf == 1 && ftype == 00)

```
UCVTF <Sd>, <Xn>, #<fbits>
```

### 64-bit to double-precision (sf == 1 && ftype == 01)

```
UCVTF <Dd>, <Xn>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer decode_fltsize;
FPRounding rounding;

case ftype of
  when '00' decode_fltsize = 32;
  when '01' decode_fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      decode_fltsize = 16;
    else
      UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

rounding = FPRoundingMode(FPCR[]);
```

## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<fbits>	For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".  For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

## Operation

```
CheckFPEnabled64();  
  
FPCRTYPE fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
constant integer fltsize = if merge then 128 else decode_fltsize;  
bits(fltsize) fltval;  
bits(intsize) intval;  
  
intval = X[n, intsize];  
fltval = if merge then V[d, fltsize] else Zeros(fltsize);  
Elem[fltval, 0, decode_fltsize] = FixedToFP(intval, fracbits, TRUE, fpcr, rounding, decode_fltsize);  
V[d, fltsize] = fltval;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UCVTF (scalar, integer)

Unsigned integer Convert to Floating-point (scalar). This instruction converts the unsigned integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the [FPCR](#), and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	ftype	1	0	0	0	1	1	0	0	0	0	0	0	Rn				Rd						
rmode												opcode																			

### 32-bit to half-precision (sf == 0 && ftype == 11) (FEAT\_FP16)

UCVTF <Hd>, <Wn>

### 32-bit to single-precision (sf == 0 && ftype == 00)

UCVTF <Sd>, <Wn>

### 32-bit to double-precision (sf == 0 && ftype == 01)

UCVTF <Dd>, <Wn>

### 64-bit to half-precision (sf == 1 && ftype == 11) (FEAT\_FP16)

UCVTF <Hd>, <Xn>

### 64-bit to single-precision (sf == 1 && ftype == 00)

UCVTF <Sd>, <Xn>

### 64-bit to double-precision (sf == 1 && ftype == 01)

UCVTF <Dd>, <Xn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer decode_ftsize;
FPRounding rounding;

case ftype of
  when '00'
    decode_ftsize = 32;
  when '01'
    decode_ftsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      decode_ftsize = 16;
    else
      UNDEFINED;

rounding = FPRoundingMode(FPCR[]);
```



## Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

## Operation

```
CheckFPEnabled64();  
  
FPCRTYPE fpcr = FPCR[];  
boolean merge = IsMerging(fpcr);  
constant integer fltsize = if merge then 128 else decode_fltsize;  
bits(fltsize) fltval;  
bits(intsize) intval;  
  
intval = X[n, intsize];  
fltval = if merge then V[d, fltsize] else Zeros(fltsize);  
Elem[fltval, 0, decode_fltsize] = FixedToFP(intval, 0, TRUE, fpcr, rounding, decode_fltsize);  
V[d, fltsize] = fltval;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UCVTF (vector, fixed-point)

Unsigned fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the [FPCR](#), and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in [FPCR](#), the exception results in either a flag being set in [FPSR](#), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the [CPACR EL1](#), [CPTR EL2](#), and [CPTR EL3](#) registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0		1	1	1	1	1	1	0	!= 0000				immb			1	1	1	0	0	1	Rn				Rd					
U									immh																						

**UCVTF** <V><d>, <V><n>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR[]);
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	1	1	0	0	1	Rn				Rd					
U									immh																						

**UCVTF** <Vd>.<T>, <Vn>.<T>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR[]);
```

### Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
000x	RESERVED
001x	H
01xx	S
1xxx	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	x	RESERVED
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in "immh:immb":

immh	<fbits>
000x	RESERVED
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in "immh:immb":

immh	<fbits>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	RESERVED
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, fpcr, rounding, esize);
V[d, 128] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UCVTF (vector, integer)

Unsigned integer Convert to Floating-point (vector). This instruction converts each element in a vector from an unsigned integer value to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR EL1*, *CPTR EL2*, and *CPTR EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#) , [Scalar single-precision and double-precision](#) , [Vector half precision](#) and [Vector single-precision and double-precision](#)

Scalar half precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn				Rd					
U																															

UCVTF <Hd> , <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn				Rd					
U																															

UCVTF <V><d> , <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector half precision  
(FEAT\_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn				Rd					
U																															

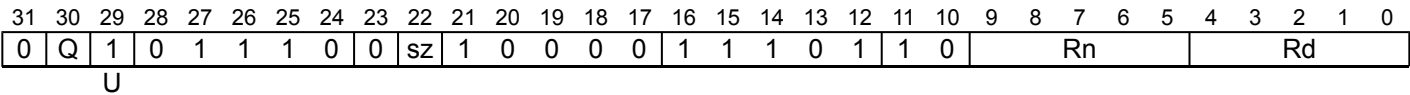
UCVTF <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



UCVTF <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

FPRounding rounding = FPRoundingMode(fpcr);
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

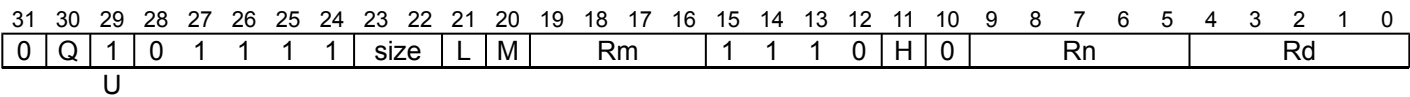
UDOT (by element)

Dot Product unsigned arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register. Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID\_AA64ISAR0\_EL1.DP indicates whether this instruction is supported.

Vector  
(FEAT\_DotProd)



UDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index = UInt(H:L);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<index> Is the element index, encoded in the "H:L" fields.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(128) operand2 = V[m, 128];
bits(datasize) result = V[d, datasize];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*index+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*index+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d, datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



UDOT (vector)

Dot Product unsigned arithmetic (vector). This instruction performs the dot product of the four unsigned 8-bit elements in each 32-bit element of the first source register with the four unsigned 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

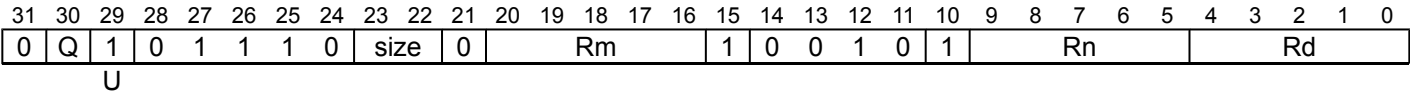
Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

*ID\_AA64ISAR0\_EL1*.DP indicates whether this instruction is supported.

Vector  
(FEAT\_DotProd)



UDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "Q":

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

result = V[d, datasize];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d, datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

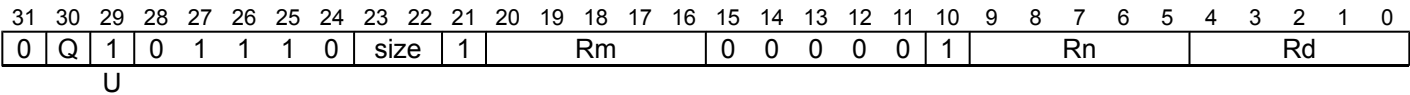
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UHADD

Unsigned Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see [URHADD](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = (element1 + element2) >> 1;
    Elem[result, e, esize] = sum<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

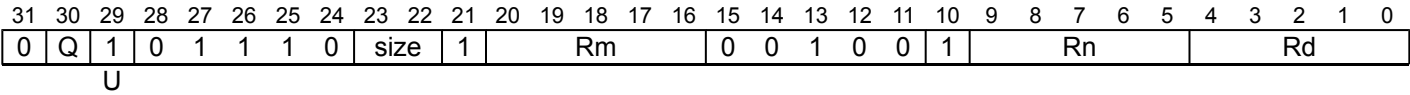
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UHSUB

Unsigned Halving Subtract. This instruction subtracts the vector elements in the second source SIMD&FP register from the corresponding vector elements in the first source SIMD&FP register, shifts each result right one bit, places each result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UHSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer diff;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = (element1 - element2) >> 1;
    Elem[result, e, esize] = diff<esize-1:0>;

V[d, datasize] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

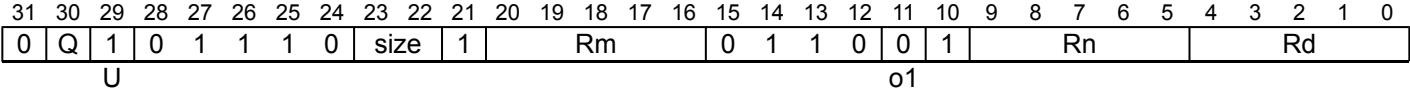
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMAX

Unsigned Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

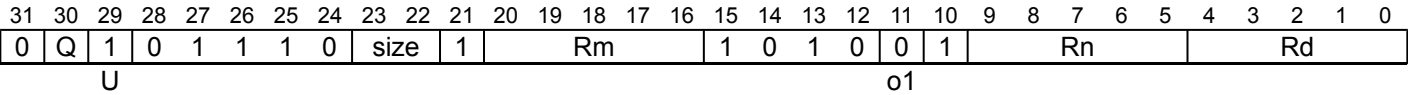
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



UMAXP

Unsigned Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



```
UMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

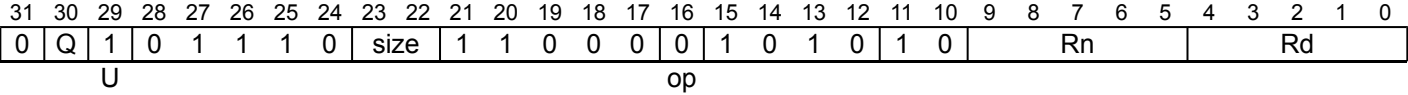
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMAXV

Unsigned Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMAXV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d, esize] = maxmin<esize-1:0>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

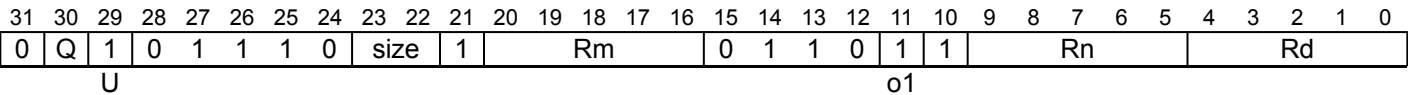
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMIN

Unsigned Minimum (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the smaller of each of the two unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

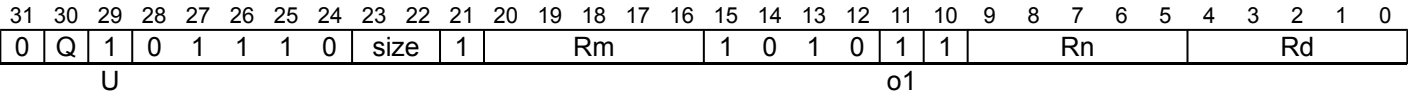
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMINP

Unsigned Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

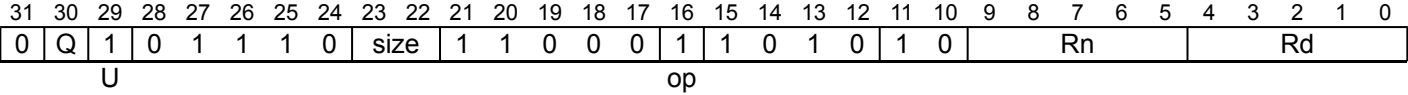
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



UMINV

Unsigned Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMINV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d, esize] = maxmin<esize-1:0>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

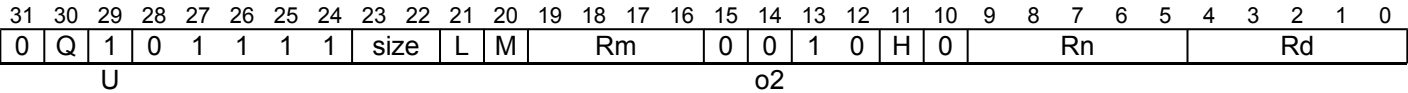
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMLAL, UMLAL2 (by element)

Unsigned Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLAL instruction extracts vector elements from the lower half of the first source register. The UMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



```
UMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxsize) operand2 = V[m, idxsize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;
V[d, 2*datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

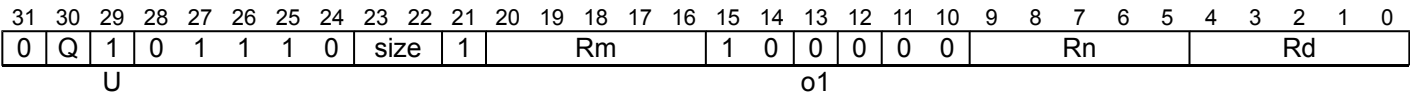


UMLAL, UMLAL2 (vector)

Unsigned Multiply-Add Long (vector). This instruction multiplies the vector elements in the lower or upper half of the first source SIMD&FP register by the corresponding vector elements of the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLAL instruction extracts vector elements from the lower half of the first source register. The UMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d, 2*datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

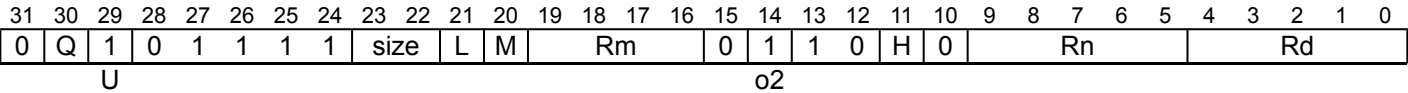
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMLSL, UMLSL2 (by element)

Unsigned Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLSL instruction extracts vector elements from the lower half of the first source register. The UMLSL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:



size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxsize) operand2 = V[m, idxsize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;
V[d, 2*datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

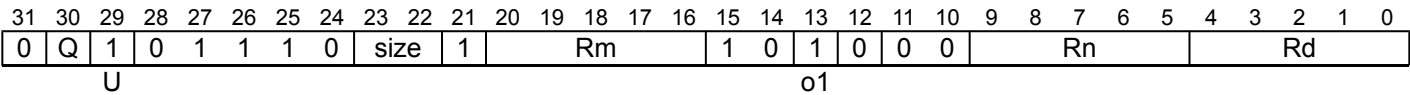


UMLSL, UMLSL2 (vector)

Unsigned Multiply-Subtract Long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The UMLSL instruction extracts each source vector from the lower half of each source register. The UMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR EL1, CPTR EL2, and CPTR EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d, 2*datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

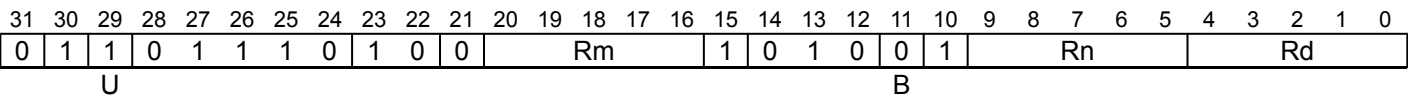
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UMMLA (vector)

Unsigned 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of unsigned 8-bit integer values in the first source vector by the 8x2 matrix of unsigned 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. [ID\\_AA64ISAR1\\_EL1](#).I8MM indicates whether this instruction is supported.

### Vector (FEAT\_I8MM)



UMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B

```
if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

### Assembler Symbols

- <Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n, 128];
bits(128) operand2 = V[m, 128];
bits(128) addend = V[d, 128];

V[d, 128] = MatMulAdd(addend, operand1, operand2, TRUE, TRUE);
```

### Operational information

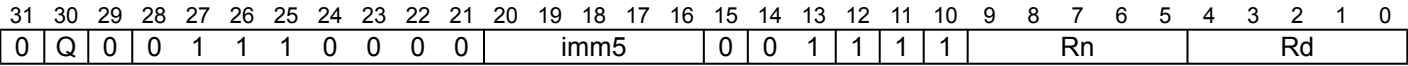
Arm expects that the UMMLA (vector) instruction will deliver a peak integer multiply throughput that is at least as high as can be achieved using two UDOT (vector) instructions, with a goal that it should have significantly higher throughput.

UMOV

Unsigned Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias [MOV \(to general\)](#).



32-bit (Q == 0)

```
UMOV <Wd>, <Vn>.<Ts>[<index>]
```

64-bit (Q == 1 && imm5 == x1000)

```
UMOV <Xd>, <Vn>.<Ts>[<index>]
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
    when '0xxxx1' size = 0;    // UMOV Wd, Vn.B
    when '0xxx10' size = 1;    // UMOV Wd, Vn.H
    when '0xx100' size = 2;    // UMOV Wd, Vn.S
    when '1x1000' size = 3;    // UMOV Xd, Vn.D
    otherwise UNDEFINED;

integer idxdsize = if imm5<4> == '1' then 128 else 64;
integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;
```

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <Ts> For the 32-bit variant: is an element size specifier, encoded in "imm5":

imm5	<Ts>
xx000	RESERVED
xxxx1	B
xxx10	H
xx100	S

For the 64-bit variant: is an element size specifier, encoded in "imm5":

imm5	<Ts>
x0000	RESERVED
xxxx1	RESERVED
xxx10	RESERVED
xx100	RESERVED
x1000	D

- <index> For the 32-bit variant: is the element index encoded in "imm5":

imm5	<index>
xx000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>

For the 64-bit variant: is the element index encoded in "imm5<4>".

## Alias Conditions

Alias	Is preferred when
<a href="#">MOV (to general)</a>	imm5 == 'x1000'
<a href="#">MOV (to general)</a>	imm5 == 'xx100'

## Operation

```

if index == 0 then
    CheckFPEEnabled64();
else
    CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n, idxdsize];

X[d, datasize] = ZeroExtend(Elem[operand, index, esize], datasize);

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

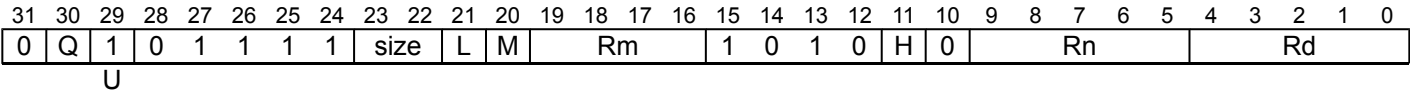
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMULL, UMULL2 (by element)

Unsigned Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMULL instruction extracts vector elements from the lower half of the first source register. The UMULL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



```
UMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
```

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:



size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxsize) operand2 = V[m, idxsize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d, 2*datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

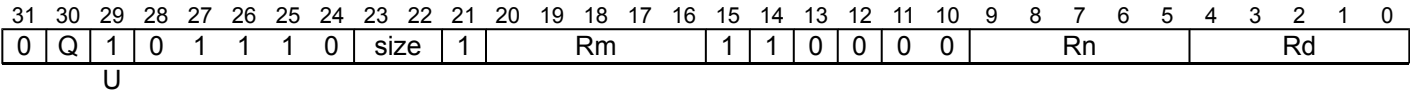
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMULL, UMULL2 (vector)

Unsigned Multiply long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The UMULL instruction extracts each source vector from the lower half of each source register. The UMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR EL1, CPTR EL2, and CPTR EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1*element2)<2*esize-1:0>;

V[d, 2*datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

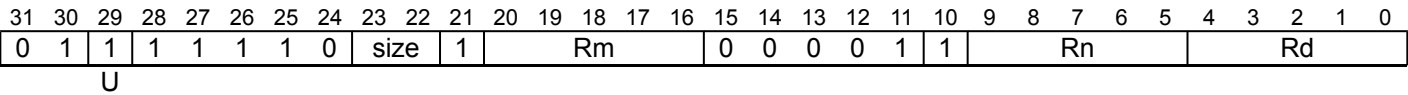
UQADD

Unsigned saturating Add. This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

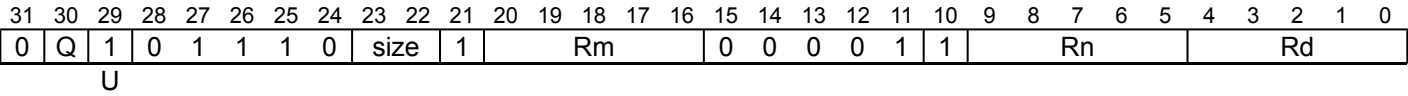
Scalar



UQADD <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector



UQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

UQRSHL

Unsigned saturating Rounding Shift Left (register). This instruction takes each vector element of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

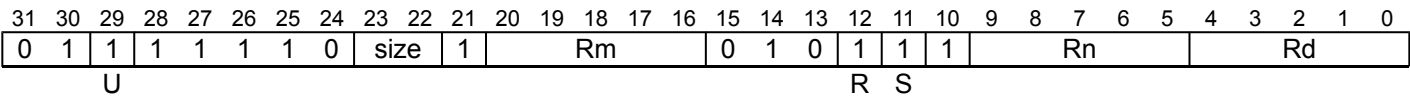
If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see [UQSHL](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

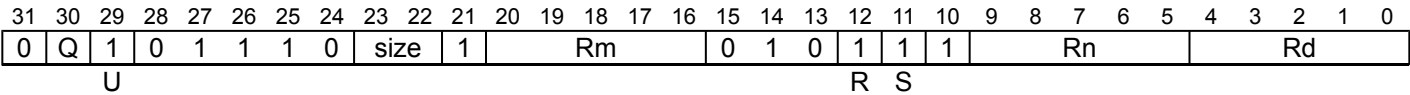
Scalar



UQRSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector



UQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);

    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UQRSHRN, UQRSHRN2

Unsigned saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see [UQSHRN](#).

The [UQRSHRN](#) instruction writes the vector to the lower half of the destination register and clears the upper half, while the [UQRSHRN2](#) instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000			immb			1	0	0	1	1	1	Rn					Rd					
U									immh									op													

**UQRSHRN** <Vb><d>, <Va><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

### Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	0	0	1	1	1	Rn					Rd				
U									immh									op													

**UQRSHRN{2}** <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```



## Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- <Tb> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- <Ta> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

- <Vb> Is the destination width specifier, encoded in "immh":

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.

- <Va> Is the source width specifier, encoded in "immh":

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

- <shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in "immh:immb":

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = RShr(Int(Elem[operand, e, 2*esize], unsigned), shift, round);
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;

```

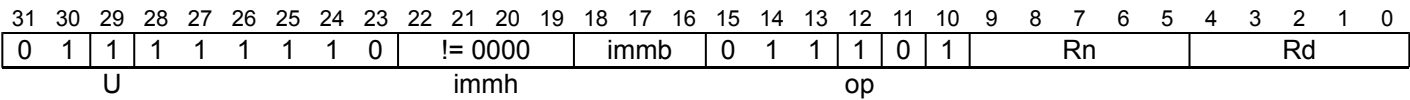
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UQSHL (immediate)

Unsigned saturating Shift Left (immediate). This instruction takes each vector element in the source SIMD&FP register, shifts it by an immediate value, places the results in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see [UQRSHL](#).  
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.  
Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.  
It has encodings from 2 classes: [Scalar](#) and [Vector](#)

### Scalar



UQSHL <v><d>, <v><n>, #<shift>

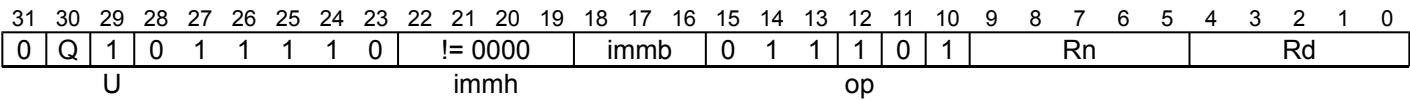
```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

### Vector



UQSHL <vd>.<T>, <vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(UInt (immh:immb)-8)
001x	(UInt (immh:immb)-16)
01xx	(UInt (immh:immb)-32)
1xxx	(UInt (immh:immb)-64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(UInt (immh:immb)-8)
001x	(UInt (immh:immb)-16)
01xx	(UInt (immh:immb)-32)
1xxx	(UInt (immh:immb)-64)

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```



## UQSHL (register)

Unsigned saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

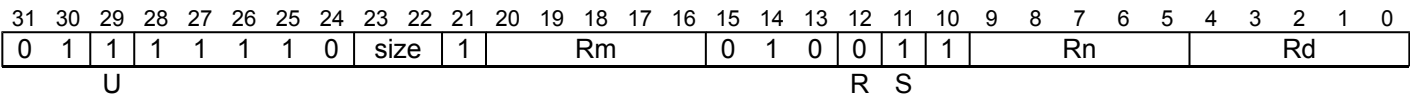
If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see [UQRSHL](#).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

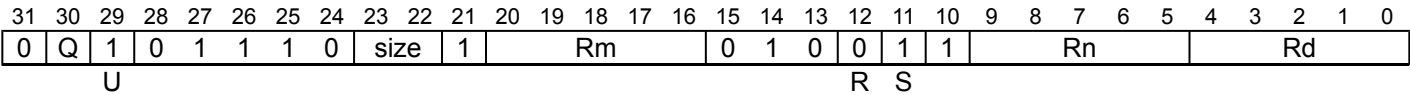
### Scalar



UQSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

### Vector



UQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

### Assembler Symbols

<V>            Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);

    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# UQSHRN, UQSHRN2

Unsigned saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see [UQSRSHRN](#).

The UQSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit [FPSR.QC](#) is set.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

## Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000			immb			1	0	0	1	0	1	Rn					Rd					
U									immh									op													

UQSHRN <Vb><d>, <Va><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

## Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			1	0	0	1	0	1	Rn					Rd				
U									immh									op													

UQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```



## Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- <Tb> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- <Ta> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

- <Vb> Is the destination width specifier, encoded in "immh":

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.

- <Va> Is the source width specifier, encoded in "immh":

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

- <shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in "immh:immb":

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = RShr(Int(Elem[operand, e, 2*esize], unsigned), shift, round);
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQSUB

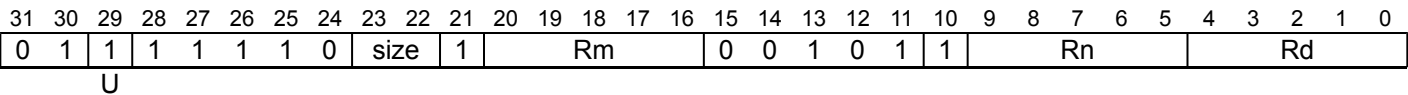
Unsigned saturating Subtract. This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit `FPSR.QC` is set.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

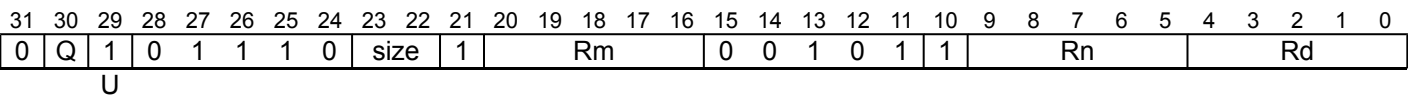
Scalar



UQSUB <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector



UQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

UQXTN, UQXTN2

Unsigned saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates each value to half the original width, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

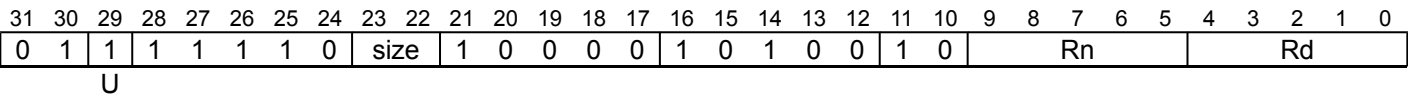
If saturation occurs, the cumulative saturation bit *FPSR.QC* is set.

The *UQXTN* instruction writes the vector to the lower half of the destination register and clears the upper half, while the *UQXTN2* instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Scalar* and *Vector*

Scalar



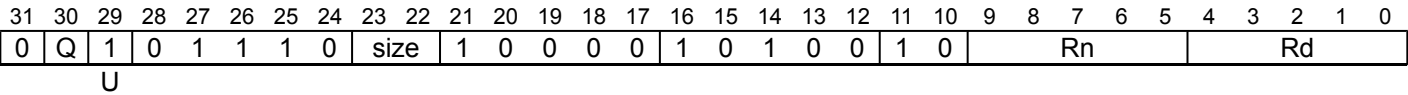
UQXTN <Vb><d>, <Va><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;

boolean unsigned = (U == '1');
```

Vector



UQXTN{2} <Vd>.<Tb>, <Vn>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q		2
0		[absent]
1		[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vb> Is the destination width specifier, encoded in “size”:

size	<Vb>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in “size”:

size	<Va>
00	H
01	S
10	D
11	RESERVED

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;
```

URECPE

Unsigned Reciprocal Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse for the unsigned integer value, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	Q	0	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn						Rd					

URECPE <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(32) element;

for e = 0 to elements-1
    element = Elem[operand, e, 32];
    Elem[result, e, 32] = UnsignedRecipEstimate(element);

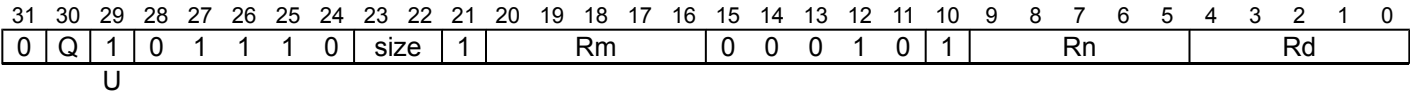
V[d, datasize] = result;
```

URHADD

Unsigned Rounding Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see [UHADD](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



URHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = (element1 + element2 + 1) >> 1;
    Elem[result, e, esize] = sum<esize-1:0>;

V[d, datasize] = result;
```





# URSHL

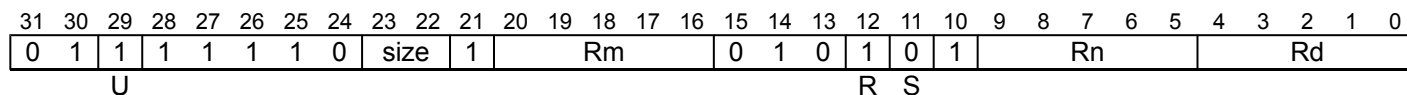
Unsigned Rounding Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

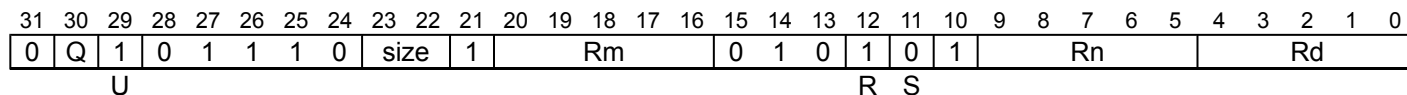
## Scalar



URSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

## Vector



URSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

## Assembler Symbols

**<V>** Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

**<d>** Is the number of the SIMD&FP destination register, in the "Rd" field.

<n>	Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
-----	--

**<m>** Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);

    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```

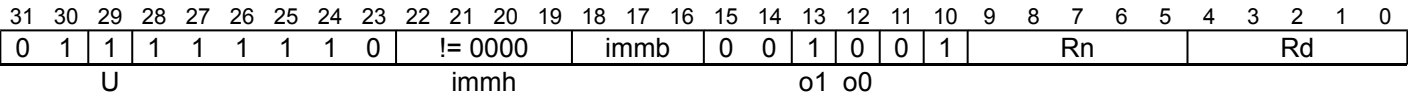
URSHR

Unsigned Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see [USHR](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar



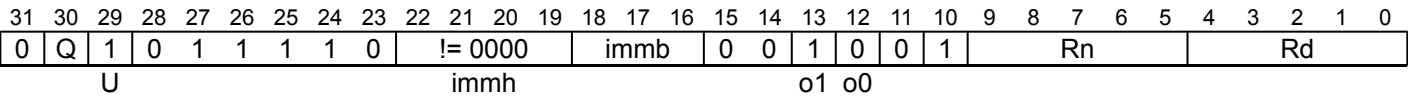
URSHR <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector



URSHR <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element = RShr(Int(Elem[operand, e, esize], unsigned), shift, round);
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d, datasize] = result;

```

URSQRTE

Unsigned Reciprocal Square Root Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse square root for each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	0	1	0	Rn					Rd				

URSQRTE <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(32) element;

for e = 0 to elements-1
    element = Elem[operand, e, 32];
    Elem[result, e, 32] = UnsignedRSqrtEstimate(element);

V[d, datasize] = result;
```

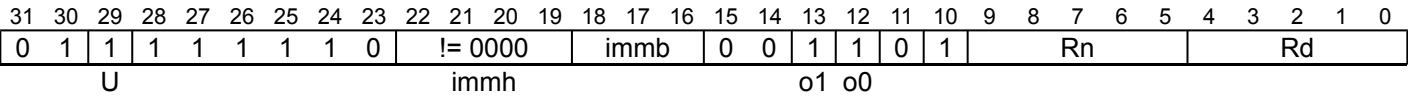
URSRA

Unsigned Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see [USRA](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar



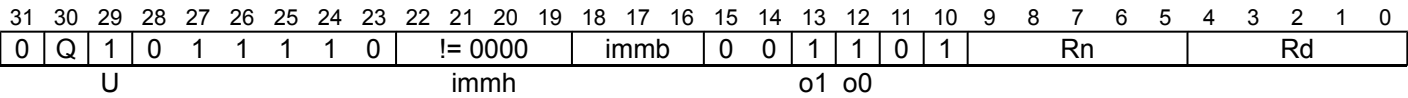
URSRA <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector



URSRA <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element = RShr(Int(Elem[operand, e, esize], unsigned), shift, round);
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d, datasize] = result;

```

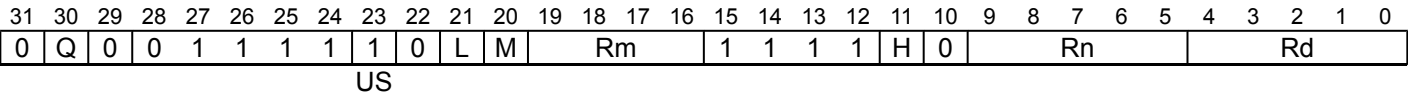


USDOT (by element)

Dot Product index form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. *ID\_AA64ISAR1\_EL1*.I8MM indicates whether this instruction is supported.

Vector  
(FEAT\_I8MM)



USDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.**4B**[<index>]

```
if !HaveInt8MatMulExt() then UNDEFINED;
boolean op1_unsigned = (US == '1');
boolean op2_unsigned = (US == '0');
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

Assembler Symbols

- <Vd>

Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
- <Ta>

Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	2S
1	4S
- <Vn>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Tb>

Is an arrangement specifier, encoded in "Q":

Q	<Tb>
0	8B
1	16B
- <Vm>

Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
- <index>

Is the immediate index of a 32-bit group of four 8-bit values in the range 0 to 3, encoded in the "H:L" fields.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(128) operand2 = V[m, 128];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;

for e = 0 to elements-1
    bits(32) res = Elem[operand3, e, 32];
    for b = 0 to 3
        integer element1 = Int(Elem[operand1, 4*e+b, 8], op1_unsigned);
        integer element2 = Int(Elem[operand2, 4*i+b, 8], op2_unsigned);
        res = res + element1 * element2;
    Elem[result, e, 32] = res;
V[d, datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USDOT (vector)

Dot Product vector form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in the corresponding 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. [ID\\_AA64ISAR1\\_EL1](#).I8MM indicates whether this instruction is supported.

Vector  
(FEAT\_I8MM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	0	0	Rm				1	0	0	1	1	1	Rn				Rd						

USDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "Q":

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;

for e = 0 to elements-1
  bits(32) res = Elem[operand3, e, 32];
  for b = 0 to 3
    integer element1 = UInt(Elem[operand1, 4*e+b, 8]);
    integer element2 = SInt(Elem[operand2, 4*e+b, 8]);
    res = res + element1 * element2;
  Elem[result, e, 32] = res;

V[d, datasize] = result;
```



USHL

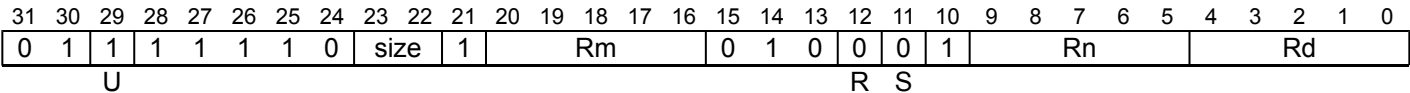
Unsigned Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see [URSHL](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

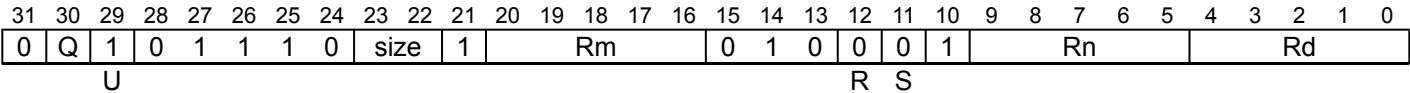
Scalar



USHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector



USHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);

    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;

```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

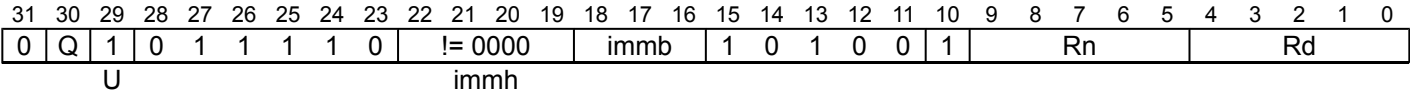
USHLL, USHLL2

Unsigned Shift Left Long (immediate). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, shifts the unsigned integer value left by the specified number of bits, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The USHLL instruction extracts vector elements from the lower half of the source register. The USHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR EL1, CPTR EL2, and CPTR EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias UXTL, UXTL2.



USHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimh);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<shift> Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(UInt (immh:immb)-8)
001x	(UInt (immh:immb)-16)
01xx	(UInt (immh:immb)-32)
1xxx	RESERVED

## Alias Conditions

Alias	Is preferred when
<a href="#">UXTL, UXTL2</a>	immb == '000' && <a href="#">BitCount</a> (immh) == 1

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part, datasize];
bits(datasize*2) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d, datasize*2] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



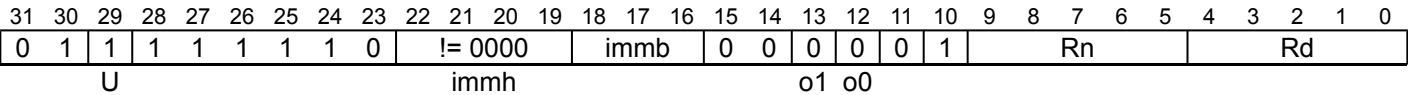
USHR

Unsigned Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see [URSHR](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar



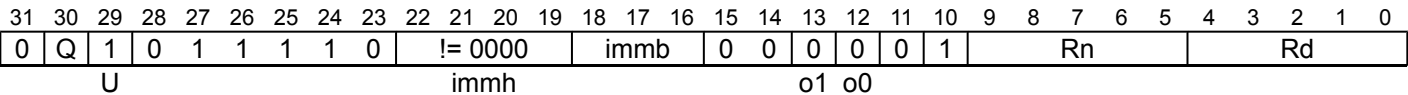
USHR <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector



USHR <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt (immh:immb) )

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt (immh:immb) )
001x	(32-UInt (immh:immb) )
01xx	(64-UInt (immh:immb) )
1xxx	(128-UInt (immh:immb) )

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element = RShr(Int(Elem[operand, e, esize], unsigned), shift, round);
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d, datasize] = result;

```

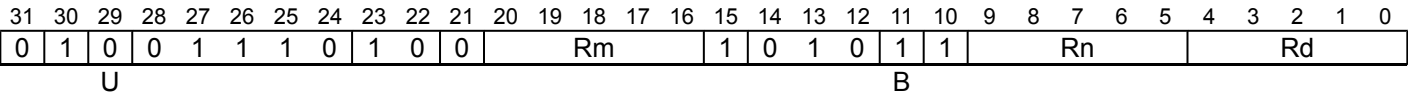
### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

USMMLA (vector)

Unsigned and signed 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of unsigned 8-bit integer values in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element. From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. *ID\_AA64ISAR1\_EL1*.I8MM indicates whether this instruction is supported.

Vector  
(FEAT\_I8MM)



USMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B

```
if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n, 128];
bits(128) operand2 = V[m, 128];
bits(128) addend = V[d, 128];

V[d, 128] = MatMulAdd(addend, operand1, operand2, TRUE, FALSE);
```

Operational information

Arm expects that the USMMLA (vector) instruction will deliver a peak integer multiply throughput that is at least as high as can be achieved using two USDOT (vector) instructions, with a goal that it should have significantly higher throughput.

USQADD

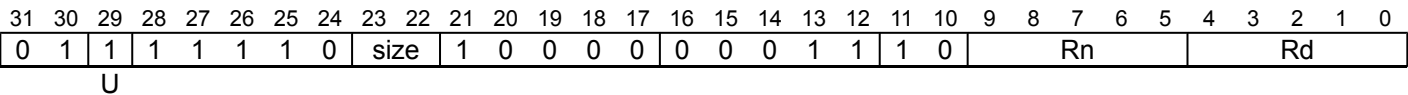
Unsigned saturating Accumulate of Signed value. This instruction adds the signed integer values of the vector elements in the source SIMD&FP register to corresponding unsigned integer values of the vector elements in the destination SIMD&FP register, and accumulates the resulting unsigned integer values with the vector elements of the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR.QC* is set.

Depending on the settings in the *CPACR\_EL1*, *CPTR\_EL2*, and *CPTR\_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar



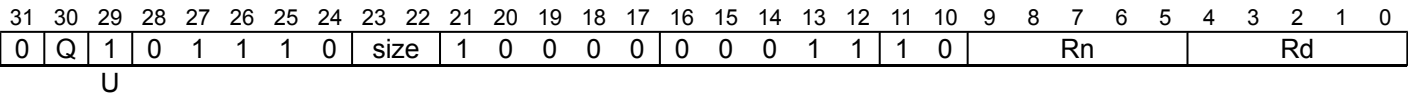
USQADD <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean unsigned = (U == '1');
```

Vector



USQADD <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

bits(datasize) operand2 = V[d, datasize];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d, datasize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

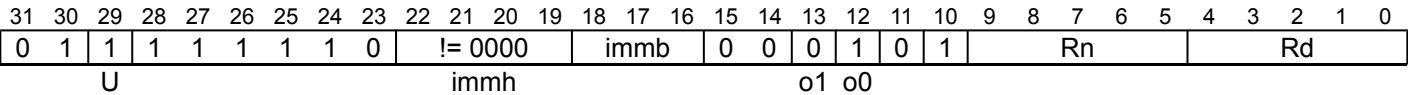
USRA

Unsigned Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see [URSRA](#).

Depending on the settings in the [CPACR\\_EL1](#), [CPTR\\_EL2](#), and [CPTR\\_EL3](#) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: [Scalar](#) and [Vector](#)

Scalar



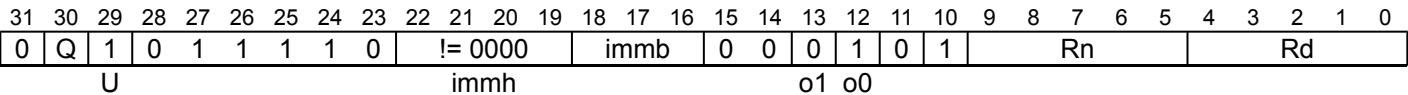
USRA <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector



USRA <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0000	x	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

- <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- <shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt (immh:immb) )

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

immh	<shift>
0000	<a href="#">SEE Advanced SIMD modified immediate</a>
0001	(16-UInt (immh:immb) )
001x	(32-UInt (immh:immb) )
01xx	(64-UInt (immh:immb) )
1xxx	(128-UInt (immh:immb) )

### Operation

```

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element = RShr(Int(Elem[operand, e, esize], unsigned), shift, round);
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d, datasize] = result;

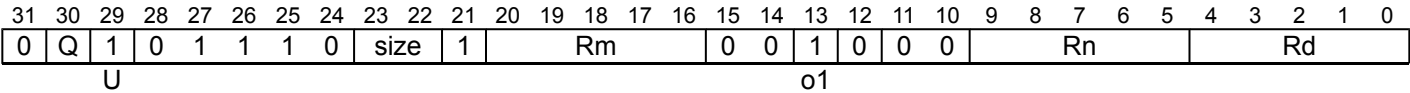
```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

USUBL, USUBL2

Unsigned Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The destination vector elements are twice as long as the source vector elements. The USUBL instruction extracts each source vector from the lower half of each source register. The USUBL2 instruction extracts each source vector from the upper half of each source register. Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



USUBL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.



## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

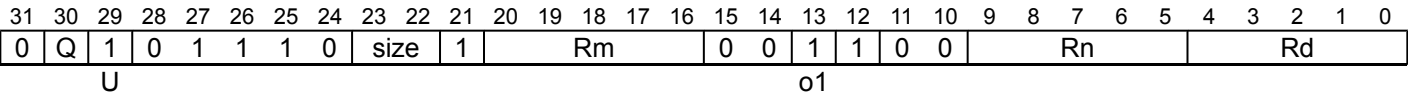
USUBW, USUBW2

Unsigned Subtract Wide. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element in the lower or upper half of the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are unsigned integer values.

The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register.

The USUBW instruction extracts vector elements from the lower half of the first source register. The USUBW2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



```
USUBW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

## Operation

```

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UXTL, UXTL2

Unsigned extend Long. This instruction copies each vector element from the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The UXTL instruction extracts vector elements from the lower half of the source register. The UXTL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of USHLL, USHLL2. This means:

- The encodings in this description are named to match the encodings of USHLL, USHLL2.
- The description of USHLL, USHLL2 gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				0	0	0	1	0	1	0	0	1	Rn				Rd					
U									immh					immb																	

UXTL{2} <Vd>.<Ta>, <Vn>.<Tb>

is equivalent to

USHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #0

and is the preferred disassembly when BitCount(immh) == 1.

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

## Operation

The description of [USHLL](#), [USHLL2](#) gives the operational pseudocode for this instruction.

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

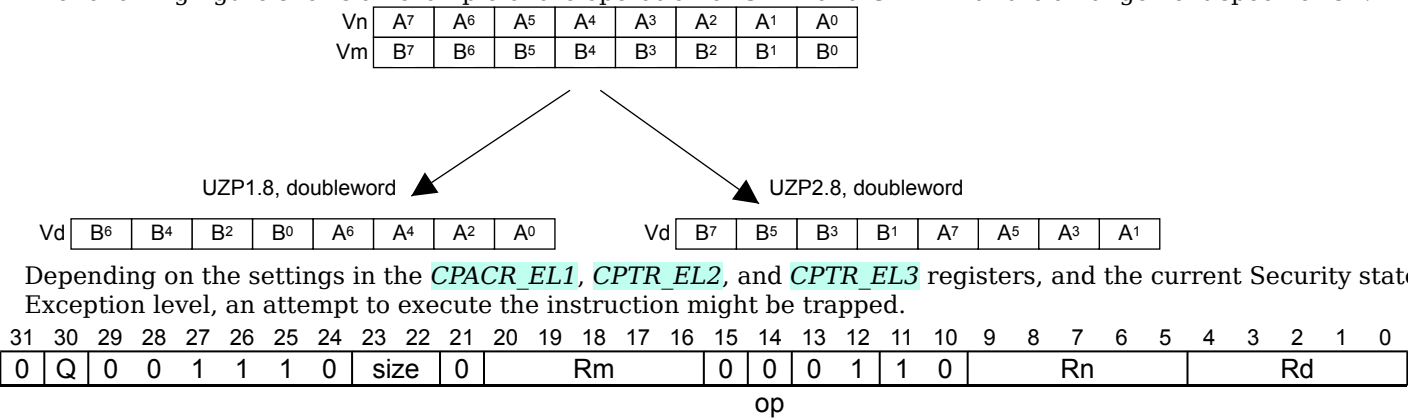
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UZP1

Unzip vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

Note

This instruction can be used with UZP2 to de-interleave two vectors.  
The following figure shows an example of the operation of UZP1 and UZP2 with the arrangement specifier 8B.



UZP1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n, datasize];
bits(datasize) operandh = V[m, datasize];
bits(datasize) result;

bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

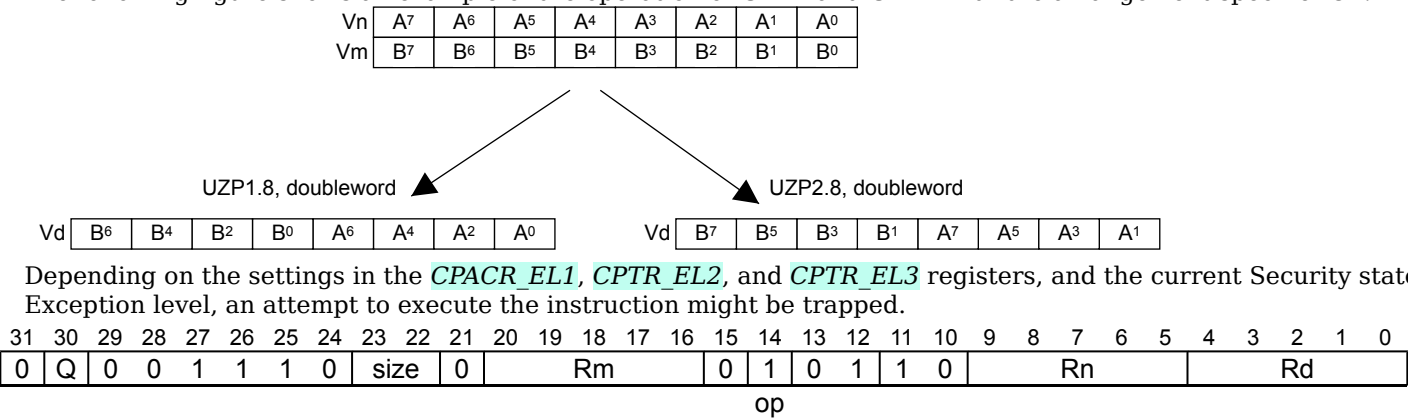
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UZP2

Unzip vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

Note

This instruction can be used with UZP1 to de-interleave two vectors.  
The following figure shows an example of the operation of UZP1 and UZP2 with the arrangement specifier 8B.



UZP2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.



## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n, datasize];
bits(datasize) operandh = V[m, datasize];
bits(datasize) result;

bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

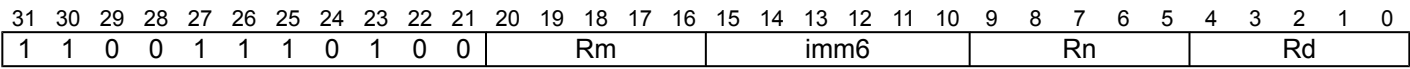
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

XAR

Exclusive-OR and Rotate performs a bitwise exclusive-OR of the 128-bit vectors in the two source SIMD&FP registers, rotates each 64-bit element of the resulting 128-bit vector right by the value specified by a 6-bit immediate value, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when **FEAT\_SHA3** is implemented.

Advanced SIMD  
(FEAT\_SHA3)



XAR <Vd>.2D, <Vn>.2D, <Vm>.2D, #<imm6>

```
if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- <imm6> Is a rotation right, encoded in "imm6".

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) tmp;
tmp = Vn EOR Vm;
V[d, 128] = ROR(tmp<127:64>, UInt(imm6)):ROR(tmp<63:0>, UInt(imm6));
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

XTN, XTN2

Extract Narrow. This instruction reads each vector element from the source SIMD&FP register, narrows each value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The `XTN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `XTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	1	0	0	0	0	1	0	0	1	0	1	0	Rn					Rd					

`XTN{2} <Vd>.<Tb>, <Vn>.<Ta>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q 2	
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;
bits(2*esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    Elem[result, e, esize] = element<esize-1:0>;
Vpart[d, part, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

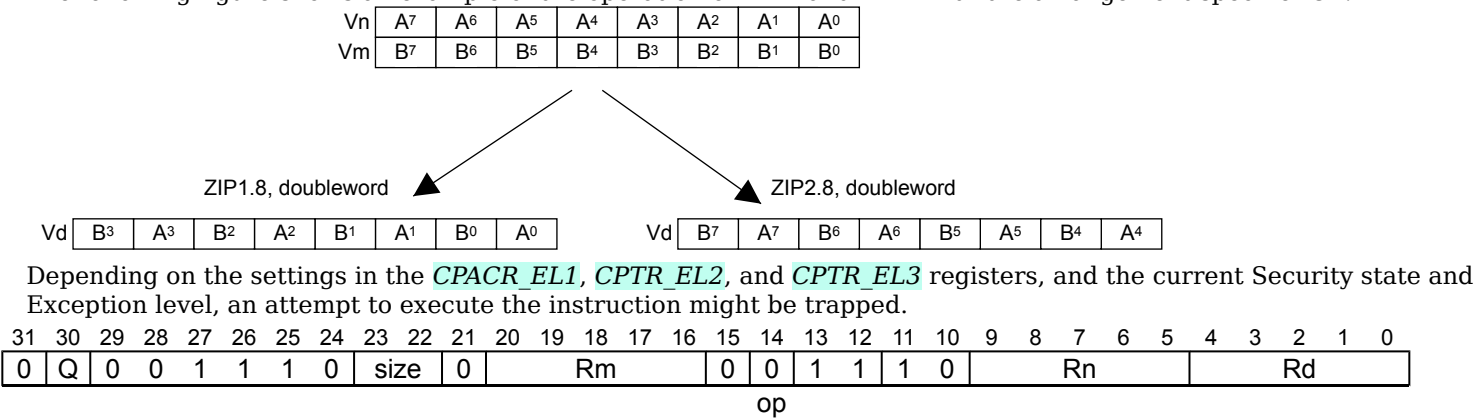
ZIP1

Zip vectors (primary). This instruction reads adjacent vector elements from the lower half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

**Note**

This instruction can be used with ZIP2 to interleave two vectors.

The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.



```
ZIP1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

integer base = part * pairs;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

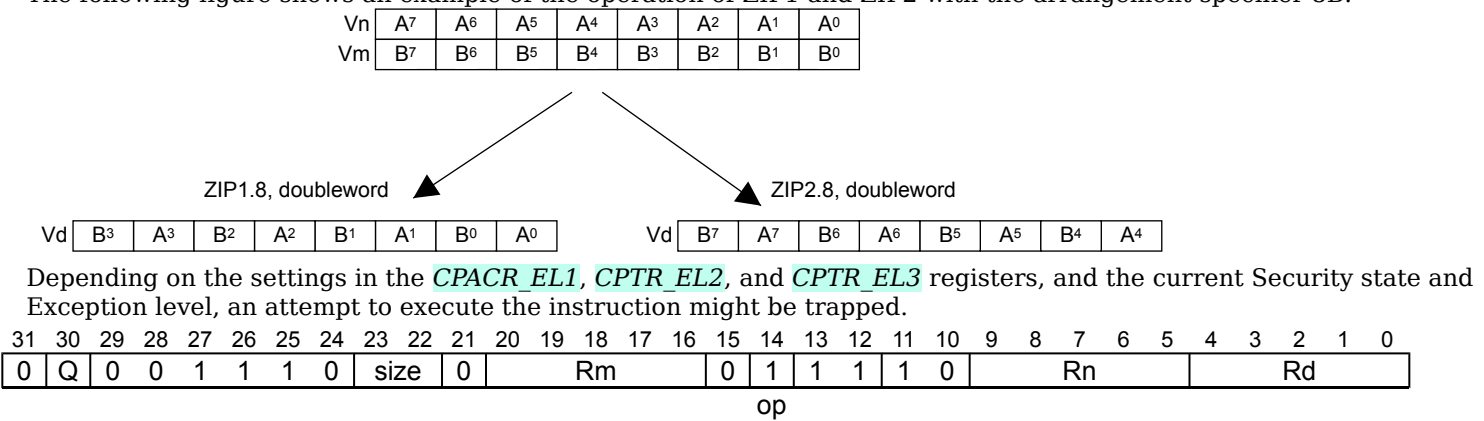
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ZIP2

Zip vectors (secondary). This instruction reads adjacent vector elements from the upper half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

Note

This instruction can be used with ZIP1 to interleave two vectors. The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.



ZIP2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

## Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

integer base = part * pairs;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

V[d, datasize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## A64 -- SVE Instructions (alphabetic order)

[ABS](#): Absolute value (predicated).

[ADCLB](#): Add with carry long (bottom).

[ADCLT](#): Add with carry long (top).

[ADD \(immediate\)](#): Add immediate (unpredicated).

[ADD \(vectors, predicated\)](#): Add vectors (predicated).

[ADD \(vectors, unpredicated\)](#): Add vectors (unpredicated).

[ADDHNB](#): Add narrow high part (bottom).

[ADDHNT](#): Add narrow high part (top).

[ADDP](#): Add pairwise.

[ADDPL](#): Add multiple of predicate register size to scalar register.

[ADDQV](#): Unsigned add reduction of quadword vector segments.

[ADDVL](#): Add multiple of vector register size to scalar register.

[ADR](#): Compute vector address.

[AESD](#): AES single round decryption.

[AESE](#): AES single round encryption.

[AESIMC](#): AES inverse mix columns.

[AESMC](#): AES mix columns.

[AND \(immediate\)](#): Bitwise AND with immediate (unpredicated).

[AND \(predicates\)](#): Bitwise AND predicates.

[AND \(vectors, predicated\)](#): Bitwise AND vectors (predicated).

[AND \(vectors, unpredicated\)](#): Bitwise AND vectors (unpredicated).

[ANDQV](#): Bitwise AND reduction of quadword vector segments.

[ANDS](#): Bitwise AND predicates, setting the condition flags.

[ANDV](#): Bitwise AND reduction to scalar.

[ASR \(immediate, predicated\)](#): Arithmetic shift right by immediate (predicated).

[ASR \(immediate, unpredicated\)](#): Arithmetic shift right by immediate (unpredicated).

[ASR \(vectors\)](#): Arithmetic shift right by vector (predicated).

[ASR \(wide elements, predicated\)](#): Arithmetic shift right by 64-bit wide elements (predicated).

[ASR \(wide elements, unpredicated\)](#): Arithmetic shift right by 64-bit wide elements (unpredicated).

[ASRD](#): Arithmetic shift right for divide by immediate (predicated).

[ASRR](#): Reversed arithmetic shift right by vector (predicated).

[BCAX](#): Bitwise clear and exclusive OR.

[BDEP](#): Scatter lower bits into positions selected by bitmask.

[BEXT](#): Gather lower bits from positions selected by bitmask.

[BFADD \(predicated\)](#): BFloat16 floating-point add vectors (predicated).

[BFADD \(unpredicated\)](#): BFloat16 floating-point add vectors (unpredicated).

[BFCLAMP](#): BFloat16 floating-point clamp to minimum/maximum number.

[BFCVT](#): Floating-point down convert to BFloat16 format (predicated).

[BFCVTNT](#): Floating-point down convert and narrow to BFloat16 (top, predicated).

[BFDOT \(indexed\)](#): BFloat16 floating-point indexed dot product.

[BFDOT \(vectors\)](#): BFloat16 floating-point dot product.

[BFMAX](#): BFloat16 floating-point maximum (predicated).

[BFMAXNM](#): BFloat16 floating-point maximum number (predicated).

[BFMIN](#): BFloat16 floating-point minimum (predicated).

[BFMINNM](#): BFloat16 floating-point minimum number (predicated).

[BFMLA \(indexed\)](#): BFloat16 floating-point fused multiply-add vectors by indexed elements.

[BFMLA \(vectors\)](#): BFloat16 floating-point fused multiply-add vectors.

[BFMLALB \(indexed\)](#): BFloat16 floating-point multiply-add long to single-precision (bottom, indexed).

[BFMLALB \(vectors\)](#): BFloat16 floating-point multiply-add long to single-precision (bottom).

[BFMLALT \(indexed\)](#): BFloat16 floating-point multiply-add long to single-precision (top, indexed).

[BFMLALT \(vectors\)](#): BFloat16 floating-point multiply-add long to single-precision (top).

[BFMLS \(indexed\)](#): BFloat16 floating-point fused multiply-subtract vectors by indexed elements.

[BFMLS \(vectors\)](#): BFloat16 floating-point fused multiply-subtract vectors.

[BFMLSLB \(indexed\)](#): BFloat16 floating-point multiply-subtract long from single-precision (bottom, indexed).

[BFMLSLB \(vectors\)](#): BFloat16 floating-point multiply-subtract long from single-precision (bottom).

[BFMLSLT \(indexed\)](#): BFloat16 floating-point multiply-subtract long from single-precision (top, indexed).

[BFMLSLT \(vectors\)](#): BFloat16 floating-point multiply-subtract long from single-precision (top).

[BFMMLA](#): BFloat16 floating-point matrix multiply-accumulate into 2×2 matrices.

[BFMUL \(indexed\)](#): BFloat16 floating-point multiply vectors by indexed elements.

[BFMUL \(vectors, predicated\)](#): BFloat16 floating-point multiply vectors (predicated).

[BFMUL \(vectors, unpredicated\)](#): BFloat16 floating-point multiply vectors (unpredicated).

[BFSUB \(predicated\)](#): BFloat16 floating-point subtract vectors (predicated).

[BFSUB \(unpredicated\)](#): BFloat16 floating-point subtract vectors (unpredicated).

[BGRP](#): Group bits to right or left as selected by bitmask.

[BIC \(immediate\)](#): Bitwise clear bits using immediate (unpredicated): an alias of AND (immediate).

[BIC \(predicates\)](#): Bitwise clear predicates.

[BIC \(vectors, predicated\)](#): Bitwise clear vectors (predicated).

[BIC \(vectors, unpredicated\)](#): Bitwise clear vectors (unpredicated).

[BICS](#): Bitwise clear predicates, setting the condition flags.

[BRKA](#): Break after first true condition.

[BRKAS](#): Break after first true condition, setting the condition flags.

[BRKB](#): Break before first true condition.

[BRKBS](#): Break before first true condition, setting the condition flags.

[BRKN](#): Propagate break to next partition.

[BRKNS](#): Propagate break to next partition, setting the condition flags.

[BRKPA](#): Break after first true condition, propagating from previous partition.

[BRKPAS](#): Break after first true condition, propagating from previous partition and setting the condition flags.

[BRKPB](#): Break before first true condition, propagating from previous partition.

[BRKPBS](#): Break before first true condition, propagating from previous partition and setting the condition flags.

[BSL](#): Bitwise select.

[BSL1N](#): Bitwise select with first input inverted.

[BSL2N](#): Bitwise select with second input inverted.

[CADD](#): Complex integer add with rotate.

[CDOT \(indexed\)](#): Complex integer dot product (indexed).

[CDOT \(vectors\)](#): Complex integer dot product.

[CLASTA \(scalar\)](#): Conditionally extract element after last to general-purpose register.

[CLASTA \(SIMD&FP scalar\)](#): Conditionally extract element after last to SIMD&FP scalar register.

[CLASTA \(vectors\)](#): Conditionally extract element after last to vector register.

[CLASTB \(scalar\)](#): Conditionally extract last element to general-purpose register.

[CLASTB \(SIMD&FP scalar\)](#): Conditionally extract last element to SIMD&FP scalar register.

[CLASTB \(vectors\)](#): Conditionally extract last element to vector register.

[CLS](#): Count leading sign bits (predicated).

[CLZ](#): Count leading zero bits (predicated).

[CMLA \(indexed\)](#): Complex integer multiply-add with rotate (indexed).

[CMLA \(vectors\)](#): Complex integer multiply-add with rotate.

[CMP<cc> \(immediate\)](#): Compare vector to immediate.

[CMP<cc> \(vectors\)](#): Compare vectors.

[CMP<cc> \(wide elements\)](#): Compare vector to 64-bit wide elements.

[CMPLE \(vectors\)](#): Compare signed less than or equal to vector, setting the condition flags: an alias of CMP<cc> (vectors).

[CMPLO \(vectors\)](#): Compare unsigned lower than vector, setting the condition flags: an alias of CMP<cc> (vectors).

[CMPLS \(vectors\)](#): Compare unsigned lower or same as vector, setting the condition flags: an alias of CMP<cc> (vectors).

[CMPLT \(vectors\)](#): Compare signed less than vector, setting the condition flags: an alias of CMP<cc> (vectors).

[CNOT](#): Logically invert boolean condition in vector (predicated).

[CNT](#): Count non-zero bits (predicated).

[CNTB, CNTD, CNTH, CNTW](#): Set scalar to multiple of predicate constraint element count.

[CNTP \(predicate as counter\)](#): Set scalar to count from predicate-as-counter.

[CNTP \(predicate\)](#): Set scalar to count of true predicate elements.

[COMPACT](#): Shuffle active elements of vector to the right and fill with zero.

[CPY \(immediate, merging\)](#): Copy signed integer immediate to vector elements (merging).

[CPY \(immediate, zeroing\)](#): Copy signed integer immediate to vector elements (zeroing).

[CPY \(scalar\)](#): Copy general-purpose register to vector elements (predicated).

[CPY \(SIMD&FP scalar\)](#): Copy SIMD&FP scalar register to vector elements (predicated).

[CTERMEQ, CTERMNE](#): Compare and terminate loop.

[DECB, DECD, DECH, DECW \(scalar\)](#): Decrement scalar by multiple of predicate constraint element count.

[DECD, DECH, DECW \(vector\)](#): Decrement vector by multiple of predicate constraint element count.

[DECP \(scalar\)](#): Decrement scalar by count of true predicate elements.

[DECP \(vector\)](#): Decrement vector by count of true predicate elements.

[DUP \(immediate\)](#): Broadcast signed immediate to vector elements (unpredicated).

[DUP \(indexed\)](#): Broadcast indexed element to vector (unpredicated).

[DUP \(scalar\)](#): Broadcast general-purpose register to vector elements (unpredicated).

[DUPM](#): Broadcast logical bitmask immediate to vector (unpredicated).

[DUPQ](#): Broadcast indexed element within each quadword vector segment (unpredicated).

[EON](#): Bitwise exclusive OR with inverted immediate (unpredicated); an alias of EOR (immediate).

[EOR \(immediate\)](#): Bitwise exclusive OR with immediate (unpredicated).

[EOR \(predicates\)](#): Bitwise exclusive OR predicates.

[EOR \(vectors, predicated\)](#): Bitwise exclusive OR vectors (predicated).

[EOR \(vectors, unpredicated\)](#): Bitwise exclusive OR vectors (unpredicated).

[EOR3](#): Bitwise exclusive OR of three vectors.

[EORBT](#): Interleaving exclusive OR (bottom, top).

[EORQV](#): Bitwise exclusive OR reduction of quadword vector segments.

[EORS](#): Bitwise exclusive OR predicates, setting the condition flags.

[EORTB](#): Interleaving exclusive OR (top, bottom).

[EORV](#): Bitwise exclusive OR reduction to scalar.

[EXT](#): Extract vector from pair of vectors.

[EXTQ](#): Extract vector segment from each pair of quadword vector segments.

[FABD](#): Floating-point absolute difference (predicated).

[FABS](#): Floating-point absolute value (predicated).

[FAC<cc>](#): Floating-point absolute compare vectors.

[FACLE](#): Floating-point absolute compare less than or equal: an alias of FAC<cc>.

[FACLT](#): Floating-point absolute compare less than: an alias of FAC<cc>.

[FADD \(immediate\)](#): Floating-point add immediate (predicated).

[FADD \(vectors, predicated\)](#): Floating-point add vector (predicated).

[FADD \(vectors, unpredicated\)](#): Floating-point add vector (unpredicated).

[FADDA](#): Floating-point add strictly-ordered reduction, accumulating in scalar.

[FADDP](#): Floating-point add pairwise.

[FADDQV](#): Floating-point add recursive reduction of quadword vector segments.

[FADDV](#): Floating-point add recursive reduction to scalar.

[FCADD](#): Floating-point complex add with rotate (predicated).

[FCLAMP](#): Floating-point clamp to minimum/maximum number.

[FCM<cc> \(vectors\)](#): Floating-point compare vectors.

[FCM<cc> \(zero\)](#): Floating-point compare vector with zero.

[FCMLA \(indexed\)](#): Floating-point complex multiply-add by indexed values with rotate.

[FCMLA \(vectors\)](#): Floating-point complex multiply-add with rotate (predicated).

[FCMLE \(vectors\)](#): Floating-point compare less than or equal to vector: an alias of FCM<cc> (vectors).

[FCMLT \(vectors\)](#): Floating-point compare less than vector: an alias of FCM<cc> (vectors).

[FCPY](#): Copy 8-bit floating-point immediate to vector elements (predicated).

[FCVT](#): Floating-point convert precision (predicated).

[FCVTLT](#): Floating-point up convert long (top, predicated).

[FCVTNT](#): Floating-point down convert and narrow (top, predicated).

[FCVTX](#): Floating-point down convert, rounding to odd (predicated).

[FCVTXNT](#): Floating-point down convert, rounding to odd (top, predicated).

[FCVTZS](#): Floating-point convert to signed integer, rounding toward zero (predicated).

[FCVTZU](#): Floating-point convert to unsigned integer, rounding toward zero (predicated).

[FDIV](#): Floating-point divide by vector (predicated).

[FDIVR](#): Floating-point reversed divide by vector (predicated).

[FDOT \(indexed\)](#): Half-precision floating-point indexed dot product.

[FDOT \(vectors\)](#): Half-precision floating-point dot product.

[FDUP](#): Broadcast 8-bit floating-point immediate to vector elements (unpredicated).

[FEXPA](#): Floating-point exponential accelerator.

[FLOGB](#): Floating-point base 2 logarithm as integer.

[FMAD](#): Floating-point fused multiply-add vectors (predicated), writing multiplicand [ $Z_{dn} = Z_a + Z_{dn} * Z_m$ ].

[FMAX \(immediate\)](#): Floating-point maximum with immediate (predicated).

[FMAX \(vectors\)](#): Floating-point maximum (predicated).

[FMAXNM \(immediate\)](#): Floating-point maximum number with immediate (predicated).

[FMAXNM \(vectors\)](#): Floating-point maximum number (predicated).

[FMAXNMP](#): Floating-point maximum number pairwise.

[FMAXNMQV](#): Floating-point maximum number recursive reduction of quadword vector segments.

[FMAXNMV](#): Floating-point maximum number recursive reduction to scalar.

[FMAXP](#): Floating-point maximum pairwise.

[FMAXQV](#): Floating-point maximum reduction of quadword vector segments.

[FMAXV](#): Floating-point maximum recursive reduction to scalar.

[FMIN \(immediate\)](#): Floating-point minimum with immediate (predicated).

[FMIN \(vectors\)](#): Floating-point minimum (predicated).

[FMINNM \(immediate\)](#): Floating-point minimum number with immediate (predicated).

[FMINNM \(vectors\)](#): Floating-point minimum number (predicated).

[FMINNMP](#): Floating-point minimum number pairwise.

[FMINNMQV](#): Floating-point minimum number recursive reduction of quadword vector segments.

[FMINNMV](#): Floating-point minimum number recursive reduction to scalar.

[FMINP](#): Floating-point minimum pairwise.

[FMINQV](#): Floating-point minimum recursive reduction of quadword vector segments.

[FMINV](#): Floating-point minimum recursive reduction to scalar.

[FMLA \(indexed\)](#): Floating-point fused multiply-add by indexed elements ( $Z_{da} = Z_{da} + Z_n * Z_m[\text{indexed}]$ ).

[FMLA \(vectors\)](#): Floating-point fused multiply-add vectors (predicated), writing addend [ $Z_{da} = Z_{da} + Z_n * Z_m$ ].

[FMLALB \(indexed\)](#): Half-precision floating-point multiply-add long to single-precision (bottom, indexed).

[FMLALB \(vectors\)](#): Half-precision floating-point multiply-add long to single-precision (bottom).

[FMLALT \(indexed\)](#): Half-precision floating-point multiply-add long to single-precision (top, indexed).

[FMLALT \(vectors\)](#): Half-precision floating-point multiply-add long to single-precision (top).

[FMLS \(indexed\)](#): Floating-point fused multiply-subtract by indexed elements ( $Z_{da} = Z_{da} + -Z_n * Z_m[\text{indexed}]$ ).

[FMLS \(vectors\)](#): Floating-point fused multiply-subtract vectors (predicated), writing addend [ $Z_{da} = Z_{da} + -Z_n * Z_m$ ].

[FMLS LB \(indexed\)](#): Half-precision floating-point multiply-subtract long from single-precision (bottom, indexed).

[FMLS LB \(vectors\)](#): Half-precision floating-point multiply-subtract long from single-precision (bottom).

[FMLS LT \(indexed\)](#): Half-precision floating-point multiply-subtract long from single-precision (top, indexed).

[FMLS LT \(vectors\)](#): Half-precision floating-point multiply-subtract long from single-precision (top).

[FMMLA](#): Floating-point matrix multiply-accumulate.

[FMOV \(immediate, predicated\)](#): Move 8-bit floating-point immediate to vector elements (predicated): an alias of FCPY.

[FMOV \(immediate, unpredicated\)](#): Move 8-bit floating-point immediate to vector elements (unpredicated): an alias of FDUP.

[FMOV \(zero, predicated\)](#): Move floating-point +0.0 to vector elements (predicated): an alias of CPY (immediate, merging).

[FMOV \(zero, unpredicated\)](#): Move floating-point +0.0 to vector elements (unpredicated): an alias of DUP (immediate).

[FMSB](#): Floating-point fused multiply-subtract vectors (predicated), writing multiplicand [ $Z_{dn} = Z_a + -Z_{dn} * Z_m$ ].

[FMUL \(immediate\)](#): Floating-point multiply by immediate (predicated).

[FMUL \(indexed\)](#): Floating-point multiply by indexed elements.

[FMUL \(vectors, predicated\)](#): Floating-point multiply vectors (predicated).

[FMUL \(vectors, unpredicated\)](#): Floating-point multiply vectors (unpredicated).

[FMULX](#): Floating-point multiply-extended vectors (predicated).

[FNEG](#): Floating-point negate (predicated).

[FNMAD](#): Floating-point negated fused multiply-add vectors (predicated), writing multiplicand [ $Z_{dn} = -Z_a + -Z_{dn} * Z_m$ ].

[FNMLA](#): Floating-point negated fused multiply-add vectors (predicated), writing addend [ $Z_{da} = -Z_{da} + -Z_n * Z_m$ ].

[FNMLS](#): Floating-point negated fused multiply-subtract vectors (predicated), writing addend [ $Z_{da} = -Z_{da} + Z_n * Z_m$ ].

[FNMSB](#): Floating-point negated fused multiply-subtract vectors (predicated), writing multiplicand [ $Z_{dn} = -Z_a + Z_{dn} * Z_m$ ].

[FRECPE](#): Floating-point reciprocal estimate (unpredicated).

[FRECPS](#): Floating-point reciprocal step (unpredicated).

[FRECPSX](#): Floating-point reciprocal exponent (predicated).

[FRINT<r>](#): Floating-point round to integral value (predicated).

[FRSQRT](#): Floating-point reciprocal square root estimate (unpredicated).

[FRSQRTS](#): Floating-point reciprocal square root step (unpredicated).

[FSCALE](#): Floating-point adjust exponent by vector (predicated).

[FSQRT](#): Floating-point square root (predicated).

[FSUB \(immediate\)](#): Floating-point subtract immediate (predicated).

[FSUB \(vectors, predicated\)](#): Floating-point subtract vectors (predicated).

[FSUB \(vectors, unpredicated\)](#): Floating-point subtract vectors (unpredicated).

[FSUBR \(immediate\)](#): Floating-point reversed subtract from immediate (predicated).

[FSUBR \(vectors\)](#): Floating-point reversed subtract vectors (predicated).

[FTMAD](#): Floating-point trigonometric multiply-add coefficient.

[FTSMUL](#): Floating-point trigonometric starting value.

[FTSSEL](#): Floating-point trigonometric select coefficient.

[HISTCNT](#): Count matching elements in vector.

[HISTSEG](#): Count matching elements in vector segments.

[INCB, INCD, INCH, INCW \(scalar\)](#): Increment scalar by multiple of predicate constraint element count.

[INCD, INCH, INCW \(vector\)](#): Increment vector by multiple of predicate constraint element count.

[INCP \(scalar\)](#): Increment scalar by count of true predicate elements.

[INCP \(vector\)](#): Increment vector by count of true predicate elements.

[INDEX \(immediate, scalar\)](#): Create index starting from immediate and incremented by general-purpose register.

[INDEX \(immediates\)](#): Create index starting from and incremented by immediate.

[INDEX \(scalar, immediate\)](#): Create index starting from general-purpose register and incremented by immediate.

[INDEX \(scalars\)](#): Create index starting from and incremented by general-purpose register.

[INSR \(scalar\)](#): Insert general-purpose register in shifted vector.

[INSR \(SIMD&FP scalar\)](#): Insert SIMD&FP scalar register in shifted vector.



[LASTA \(scalar\)](#): Extract element after last to general-purpose register.

[LASTA \(SIMD&FP scalar\)](#): Extract element after last to SIMD&FP scalar register.

[LASTB \(scalar\)](#): Extract last element to general-purpose register.

[LASTB \(SIMD&FP scalar\)](#): Extract last element to SIMD&FP scalar register.

[LD1B \(scalar plus immediate, consecutive registers\)](#): Contiguous load of bytes to multiple consecutive vectors (immediate index).

[LD1B \(scalar plus immediate, single register\)](#): Contiguous load unsigned bytes to vector (immediate index).

[LD1B \(scalar plus scalar, consecutive registers\)](#): Contiguous load of bytes to multiple consecutive vectors (scalar index).

[LD1B \(scalar plus scalar, single register\)](#): Contiguous load unsigned bytes to vector (scalar index).

[LD1B \(scalar plus vector\)](#): Gather load unsigned bytes to vector (vector index).

[LD1B \(vector plus immediate\)](#): Gather load unsigned bytes to vector (immediate index).

[LD1D \(scalar plus immediate, consecutive registers\)](#): Contiguous load of doublewords to multiple consecutive vectors (immediate index).

[LD1D \(scalar plus immediate, single register\)](#): Contiguous load unsigned doublewords to vector (immediate index).

[LD1D \(scalar plus scalar, consecutive registers\)](#): Contiguous load of doublewords to multiple consecutive vectors (scalar index).

[LD1D \(scalar plus scalar, single register\)](#): Contiguous load unsigned doublewords to vector (scalar index).

[LD1D \(scalar plus vector\)](#): Gather load doublewords to vector (vector index).

[LD1D \(vector plus immediate\)](#): Gather load doublewords to vector (immediate index).

[LD1H \(scalar plus immediate, consecutive registers\)](#): Contiguous load of halfwords to multiple consecutive vectors (immediate index).

[LD1H \(scalar plus immediate, single register\)](#): Contiguous load unsigned halfwords to vector (immediate index).

[LD1H \(scalar plus scalar, consecutive registers\)](#): Contiguous load of halfwords to multiple consecutive vectors (scalar index).

[LD1H \(scalar plus scalar, single register\)](#): Contiguous load unsigned halfwords to vector (scalar index).

[LD1H \(scalar plus vector\)](#): Gather load unsigned halfwords to vector (vector index).

[LD1H \(vector plus immediate\)](#): Gather load unsigned halfwords to vector (immediate index).

[LD1Q](#): Gather load quadwords.

[LD1RB](#): Load and broadcast unsigned byte to vector.

[LD1RD](#): Load and broadcast doubleword to vector.

[LD1RH](#): Load and broadcast unsigned halfword to vector.

[LD1ROB \(scalar plus immediate\)](#): Contiguous load and replicate thirty-two bytes (immediate index).

[LD1ROB \(scalar plus scalar\)](#): Contiguous load and replicate thirty-two bytes (scalar index).

[LD1ROD \(scalar plus immediate\)](#): Contiguous load and replicate four doublewords (immediate index).

[LD1ROD \(scalar plus scalar\)](#): Contiguous load and replicate four doublewords (scalar index).

[LD1ROH \(scalar plus immediate\)](#): Contiguous load and replicate sixteen halfwords (immediate index).

[LD1ROH \(scalar plus scalar\)](#): Contiguous load and replicate sixteen halfwords (scalar index).

[LD1ROW \(scalar plus immediate\)](#): Contiguous load and replicate eight words (immediate index).



[LD1ROW \(scalar plus scalar\)](#): Contiguous load and replicate eight words (scalar index).

[LD1ROB \(scalar plus immediate\)](#): Contiguous load and replicate sixteen bytes (immediate index).

[LD1ROB \(scalar plus scalar\)](#): Contiguous load and replicate sixteen bytes (scalar index).

[LD1ROD \(scalar plus immediate\)](#): Contiguous load and replicate two doublewords (immediate index).

[LD1ROD \(scalar plus scalar\)](#): Contiguous load and replicate two doublewords (scalar index).

[LD1ROH \(scalar plus immediate\)](#): Contiguous load and replicate eight halfwords (immediate index).

[LD1ROH \(scalar plus scalar\)](#): Contiguous load and replicate eight halfwords (scalar index).

[LD1RQW \(scalar plus immediate\)](#): Contiguous load and replicate four words (immediate index).

[LD1RQW \(scalar plus scalar\)](#): Contiguous load and replicate four words (scalar index).

[LD1RSB](#): Load and broadcast signed byte to vector.

[LD1RSH](#): Load and broadcast signed halfword to vector.

[LD1RSW](#): Load and broadcast signed word to vector.

[LD1RW](#): Load and broadcast unsigned word to vector.

[LD1SB \(scalar plus immediate\)](#): Contiguous load signed bytes to vector (immediate index).

[LD1SB \(scalar plus scalar\)](#): Contiguous load signed bytes to vector (scalar index).

[LD1SB \(scalar plus vector\)](#): Gather load signed bytes to vector (vector index).

[LD1SB \(vector plus immediate\)](#): Gather load signed bytes to vector (immediate index).

[LD1SH \(scalar plus immediate\)](#): Contiguous load signed halfwords to vector (immediate index).

[LD1SH \(scalar plus scalar\)](#): Contiguous load signed halfwords to vector (scalar index).

[LD1SH \(scalar plus vector\)](#): Gather load signed halfwords to vector (vector index).

[LD1SH \(vector plus immediate\)](#): Gather load signed halfwords to vector (immediate index).

[LD1SW \(scalar plus immediate\)](#): Contiguous load signed words to vector (immediate index).

[LD1SW \(scalar plus scalar\)](#): Contiguous load signed words to vector (scalar index).

[LD1SW \(scalar plus vector\)](#): Gather load signed words to vector (vector index).

[LD1SW \(vector plus immediate\)](#): Gather load signed words to vector (immediate index).

[LD1W \(scalar plus immediate, consecutive registers\)](#): Contiguous load of words to multiple consecutive vectors (immediate index).

[LD1W \(scalar plus immediate, single register\)](#): Contiguous load unsigned words to vector (immediate index).

[LD1W \(scalar plus scalar, consecutive registers\)](#): Contiguous load of words to multiple consecutive vectors (scalar index).

[LD1W \(scalar plus scalar, single register\)](#): Contiguous load unsigned words to vector (scalar index).

[LD1W \(scalar plus vector\)](#): Gather load unsigned words to vector (vector index).

[LD1W \(vector plus immediate\)](#): Gather load unsigned words to vector (immediate index).

[LD2B \(scalar plus immediate\)](#): Contiguous load two-byte structures to two vectors (immediate index).

[LD2B \(scalar plus scalar\)](#): Contiguous load two-byte structures to two vectors (scalar index).

[LD2D \(scalar plus immediate\)](#): Contiguous load two-doubleword structures to two vectors (immediate index).

[LD2D \(scalar plus scalar\)](#): Contiguous load two-doubleword structures to two vectors (scalar index).

[LD2H \(scalar plus immediate\)](#): Contiguous load two-halfword structures to two vectors (immediate index).

[LD2H \(scalar plus scalar\)](#): Contiguous load two-halfword structures to two vectors (scalar index).

[LD2Q \(scalar plus immediate\)](#): Contiguous load two-quadword structures to two vectors (immediate index).

[LD2Q \(scalar plus scalar\)](#): Contiguous load two-quadword structures to two vectors (scalar index).

[LD2W \(scalar plus immediate\)](#): Contiguous load two-word structures to two vectors (immediate index).

[LD2W \(scalar plus scalar\)](#): Contiguous load two-word structures to two vectors (scalar index).

[LD3B \(scalar plus immediate\)](#): Contiguous load three-byte structures to three vectors (immediate index).

[LD3B \(scalar plus scalar\)](#): Contiguous load three-byte structures to three vectors (scalar index).

[LD3D \(scalar plus immediate\)](#): Contiguous load three-doubleword structures to three vectors (immediate index).

[LD3D \(scalar plus scalar\)](#): Contiguous load three-doubleword structures to three vectors (scalar index).

[LD3H \(scalar plus immediate\)](#): Contiguous load three-halfword structures to three vectors (immediate index).

[LD3H \(scalar plus scalar\)](#): Contiguous load three-halfword structures to three vectors (scalar index).

[LD3Q \(scalar plus immediate\)](#): Contiguous load three-quadword structures to three vectors (immediate index).

[LD3Q \(scalar plus scalar\)](#): Contiguous load three-quadword structures to three vectors (scalar index).

[LD3W \(scalar plus immediate\)](#): Contiguous load three-word structures to three vectors (immediate index).

[LD3W \(scalar plus scalar\)](#): Contiguous load three-word structures to three vectors (scalar index).

[LD4B \(scalar plus immediate\)](#): Contiguous load four-byte structures to four vectors (immediate index).

[LD4B \(scalar plus scalar\)](#): Contiguous load four-byte structures to four vectors (scalar index).

[LD4D \(scalar plus immediate\)](#): Contiguous load four-doubleword structures to four vectors (immediate index).

[LD4D \(scalar plus scalar\)](#): Contiguous load four-doubleword structures to four vectors (scalar index).

[LD4H \(scalar plus immediate\)](#): Contiguous load four-halfword structures to four vectors (immediate index).

[LD4H \(scalar plus scalar\)](#): Contiguous load four-halfword structures to four vectors (scalar index).

[LD4Q \(scalar plus immediate\)](#): Contiguous load four-quadword structures to four vectors (immediate index).

[LD4Q \(scalar plus scalar\)](#): Contiguous load four-quadword structures to four vectors (scalar index).

[LD4W \(scalar plus immediate\)](#): Contiguous load four-word structures to four vectors (immediate index).

[LD4W \(scalar plus scalar\)](#): Contiguous load four-word structures to four vectors (scalar index).

[LDFF1B \(scalar plus scalar\)](#): Contiguous load first-fault unsigned bytes to vector (scalar index).

[LDFF1B \(scalar plus vector\)](#): Gather load first-fault unsigned bytes to vector (vector index).

[LDFF1B \(vector plus immediate\)](#): Gather load first-fault unsigned bytes to vector (immediate index).

[LDFF1D \(scalar plus scalar\)](#): Contiguous load first-fault doublewords to vector (scalar index).

[LDFF1D \(scalar plus vector\)](#): Gather load first-fault doublewords to vector (vector index).

[LDFF1D \(vector plus immediate\)](#): Gather load first-fault doublewords to vector (immediate index).

[LDFF1H \(scalar plus scalar\)](#): Contiguous load first-fault unsigned halfwords to vector (scalar index).

[LDFF1H \(scalar plus vector\)](#): Gather load first-fault unsigned halfwords to vector (vector index).

[LDFF1H \(vector plus immediate\)](#): Gather load first-fault unsigned halfwords to vector (immediate index).

[LDFF1SB \(scalar plus scalar\)](#): Contiguous load first-fault signed bytes to vector (scalar index).

[LDFF1SB \(scalar plus vector\)](#): Gather load first-fault signed bytes to vector (vector index).

[LDFF1SB \(vector plus immediate\)](#): Gather load first-fault signed bytes to vector (immediate index).

[LDFF1SH \(scalar plus scalar\)](#): Contiguous load first-fault signed halfwords to vector (scalar index).

[LDFF1SH \(scalar plus vector\)](#): Gather load first-fault signed halfwords to vector (vector index).

[LDFF1SH \(vector plus immediate\)](#): Gather load first-fault signed halfwords to vector (immediate index).

[LDFF1SW \(scalar plus scalar\)](#): Contiguous load first-fault signed words to vector (scalar index).

[LDFF1SW \(scalar plus vector\)](#): Gather load first-fault signed words to vector (vector index).

[LDFF1SW \(vector plus immediate\)](#): Gather load first-fault signed words to vector (immediate index).

[LDFF1W \(scalar plus scalar\)](#): Contiguous load first-fault unsigned words to vector (scalar index).

[LDFF1W \(scalar plus vector\)](#): Gather load first-fault unsigned words to vector (vector index).

[LDFF1W \(vector plus immediate\)](#): Gather load first-fault unsigned words to vector (immediate index).

[LDNF1B](#): Contiguous load non-fault unsigned bytes to vector (immediate index).

[LDNF1D](#): Contiguous load non-fault doublewords to vector (immediate index).

[LDNF1H](#): Contiguous load non-fault unsigned halfwords to vector (immediate index).

[LDNF1SB](#): Contiguous load non-fault signed bytes to vector (immediate index).

[LDNF1SH](#): Contiguous load non-fault signed halfwords to vector (immediate index).

[LDNF1SW](#): Contiguous load non-fault signed words to vector (immediate index).

[LDNF1W](#): Contiguous load non-fault unsigned words to vector (immediate index).

[LDNT1B \(scalar plus immediate, consecutive registers\)](#): Contiguous load non-temporal of bytes to multiple consecutive vectors (immediate index).

[LDNT1B \(scalar plus immediate, single register\)](#): Contiguous load non-temporal bytes to vector (immediate index).

[LDNT1B \(scalar plus scalar, consecutive registers\)](#): Contiguous load non-temporal of bytes to multiple consecutive vectors (scalar index).

[LDNT1B \(scalar plus scalar, single register\)](#): Contiguous load non-temporal bytes to vector (scalar index).

[LDNT1B \(vector plus scalar\)](#): Gather load non-temporal unsigned bytes.

[LDNT1D \(scalar plus immediate, consecutive registers\)](#): Contiguous load non-temporal of doublewords to multiple consecutive vectors (immediate index).

[LDNT1D \(scalar plus immediate, single register\)](#): Contiguous load non-temporal doublewords to vector (immediate index).

[LDNT1D \(scalar plus scalar, consecutive registers\)](#): Contiguous load non-temporal of doublewords to multiple consecutive vectors (scalar index).

[LDNT1D \(scalar plus scalar, single register\)](#): Contiguous load non-temporal doublewords to vector (scalar index).

[LDNT1D \(vector plus scalar\)](#): Gather load non-temporal unsigned doublewords.

[LDNT1H \(scalar plus immediate, consecutive registers\)](#): Contiguous load non-temporal of halfwords to multiple consecutive vectors (immediate index).

[LDNT1H \(scalar plus immediate, single register\)](#): Contiguous load non-temporal halfwords to vector (immediate index).

[LDNT1H \(scalar plus scalar, consecutive registers\)](#): Contiguous load non-temporal of halfwords to multiple consecutive vectors (scalar index).

[LDNT1H \(scalar plus scalar, single register\)](#): Contiguous load non-temporal halfwords to vector (scalar index).

[LDNT1H \(vector plus scalar\)](#): Gather load non-temporal unsigned halfwords.

[LDNT1SB](#): Gather load non-temporal signed bytes.

[LDNT1SH](#): Gather load non-temporal signed halfwords.

[LDNT1SW](#): Gather load non-temporal signed words.

[LDNT1W \(scalar plus immediate, consecutive registers\)](#): Contiguous load non-temporal of words to multiple consecutive vectors (immediate index).

[LDNT1W \(scalar plus immediate, single register\)](#): Contiguous load non-temporal words to vector (immediate index).

[LDNT1W \(scalar plus scalar, consecutive registers\)](#): Contiguous load non-temporal of words to multiple consecutive vectors (scalar index).

[LDNT1W \(scalar plus scalar, single register\)](#): Contiguous load non-temporal words to vector (scalar index).

[LDNT1W \(vector plus scalar\)](#): Gather load non-temporal unsigned words.

[LDR \(predicate\)](#): Load predicate register.

[LDR \(vector\)](#): Load vector register.

[LSL \(immediate, predicated\)](#): Logical shift left by immediate (predicated).

[LSL \(immediate, unpredicated\)](#): Logical shift left by immediate (unpredicated).

[LSL \(vectors\)](#): Logical shift left by vector (predicated).

[LSL \(wide elements, predicated\)](#): Logical shift left by 64-bit wide elements (predicated).

[LSL \(wide elements, unpredicated\)](#): Logical shift left by 64-bit wide elements (unpredicated).

[LSLR](#): Reversed logical shift left by vector (predicated).

[LSR \(immediate, predicated\)](#): Logical shift right by immediate (predicated).

[LSR \(immediate, unpredicated\)](#): Logical shift right by immediate (unpredicated).

[LSR \(vectors\)](#): Logical shift right by vector (predicated).

[LSR \(wide elements, predicated\)](#): Logical shift right by 64-bit wide elements (predicated).

[LSR \(wide elements, unpredicated\)](#): Logical shift right by 64-bit wide elements (unpredicated).

[LSRR](#): Reversed logical shift right by vector (predicated).

[MAD](#): Multiply-add vectors (predicated), writing multiplicand [ $Z_{dn} = Z_a + Z_{dn} * Z_m$ ].

[MATCH](#): Detect any matching elements, setting the condition flags.

[MLA \(indexed\)](#): Multiply-add to accumulator (indexed).

[MLA \(vectors\)](#): Multiply-add vectors (predicated), writing addend [ $Z_{da} = Z_{da} + Z_n * Z_m$ ].

[MLS \(indexed\)](#): Multiply-subtract from accumulator (indexed).

[MLS \(vectors\)](#): Multiply-subtract vectors (predicated), writing addend [ $Z_{da} = Z_{da} - Z_n * Z_m$ ].

[MOV](#): Move logical bitmask immediate to vector (unpredicated): an alias of DUPM.

[MOV](#): Move predicate (unpredicated): an alias of ORR (predicates).

[MOV \(immediate, predicated, merging\)](#): Move signed integer immediate to vector elements (merging): an alias of CPY (immediate, merging).

[MOV \(immediate, predicated, zeroing\)](#): Move signed integer immediate to vector elements (zeroing): an alias of CPY (immediate, zeroing).

[MOV \(immediate, unpredicated\)](#): Move signed immediate to vector elements (unpredicated): an alias of DUP (immediate).

[MOV \(predicate, predicated, merging\)](#): Move predicates (merging): an alias of SEL (predicates).

[MOV \(predicate, predicated, zeroing\)](#): Move predicates (zeroing): an alias of AND (predicates).

[MOV \(scalar, predicated\)](#): Move general-purpose register to vector elements (predicated): an alias of CPY (scalar).

[MOV \(scalar, unpredicated\)](#): Move general-purpose register to vector elements (unpredicated): an alias of DUP (scalar).

[MOV \(SIMD&FP scalar, predicated\)](#): Move SIMD&FP scalar register to vector elements (predicated): an alias of CPY (SIMD&FP scalar).

[MOV \(SIMD&FP scalar, unpredicated\)](#): Move indexed element or SIMD&FP scalar to vector (unpredicated): an alias of DUP (indexed).

[MOV \(vector, predicated\)](#): Move vector elements (predicated): an alias of SEL (vectors).

[MOV \(vector, unpredicated\)](#): Move vector register (unpredicated): an alias of ORR (vectors, unpredicated).

[MOVPRFX \(predicated\)](#): Move prefix (predicated).

[MOVPRFX \(unpredicated\)](#): Move prefix (unpredicated).

[MOVS \(predicated\)](#): Move predicates (zeroing), setting the condition flags: an alias of ANDS.

[MOVS \(unpredicated\)](#): Move predicate (unpredicated), setting the condition flags: an alias of ORRS.

[MSB](#): Multiply-subtract vectors (predicated), writing multiplicand [ $Z_{dn} = Z_a - Z_{dn} * Z_m$ ].

[MUL \(immediate\)](#): Multiply by immediate (unpredicated).

[MUL \(indexed\)](#): Multiply (indexed).

[MUL \(vectors, predicated\)](#): Multiply vectors (predicated).

[MUL \(vectors, unpredicated\)](#): Multiply vectors (unpredicated).

[NAND](#): Bitwise NAND predicates.

[NANDS](#): Bitwise NAND predicates, setting the condition flags.

[NBSL](#): Bitwise inverted select.

[NEG](#): Negate (predicated).

[NMATCH](#): Detect no matching elements, setting the condition flags.

[NOR](#): Bitwise NOR predicates.

[NORS](#): Bitwise NOR predicates, setting the condition flags.

[NOT \(predicate\)](#): Bitwise invert predicate: an alias of EOR (predicates).

[NOT \(vector\)](#): Bitwise invert vector (predicated).

[NOTS](#): Bitwise invert predicate, setting the condition flags: an alias of EORS.

[ORN \(immediate\)](#): Bitwise inclusive OR with inverted immediate (unpredicated): an alias of ORR (immediate).

[ORN \(predicates\)](#): Bitwise inclusive OR inverted predicate.

[ORNS](#): Bitwise inclusive OR inverted predicate, setting the condition flags.

[ORQV](#): Bitwise inclusive OR reduction of quadword vector segments.

[ORR \(immediate\)](#): Bitwise inclusive OR with immediate (unpredicated).

[ORR \(predicates\)](#): Bitwise inclusive OR predicates.

[ORR \(vectors, predicated\)](#): Bitwise inclusive OR vectors (predicated).

[ORR \(vectors, unpredicated\)](#): Bitwise inclusive OR vectors (unpredicated).

[ORRS](#): Bitwise inclusive OR predicates, setting the condition flags.

[ORV](#): Bitwise inclusive OR reduction to scalar.

[PEXT \(predicate pair\)](#): Set pair of predicates from predicate-as-counter.

[PEXT \(predicate\)](#): Set predicate from predicate-as-counter.

[PFALSE](#): Set all predicate elements to false.

[PFIRST](#): Set the first active predicate element to true.

[PMOV \(to predicate\)](#): Move predicate from vector.

[PMOV \(to vector\)](#): Move predicate to vector.

[PMUL](#): Polynomial multiply vectors (unpredicated).

[PMULLB](#): Polynomial multiply long (bottom).

[PMULLT](#): Polynomial multiply long (top).

[PNEXT](#): Find next active predicate.

[PRFB \(scalar plus immediate\)](#): Contiguous prefetch bytes (immediate index).

[PRFB \(scalar plus scalar\)](#): Contiguous prefetch bytes (scalar index).

[PRFB \(scalar plus vector\)](#): Gather prefetch bytes (scalar plus vector).

[PRFB \(vector plus immediate\)](#): Gather prefetch bytes (vector plus immediate).

[PRFD \(scalar plus immediate\)](#): Contiguous prefetch doublewords (immediate index).

[PRFD \(scalar plus scalar\)](#): Contiguous prefetch doublewords (scalar index).

[PRFD \(scalar plus vector\)](#): Gather prefetch doublewords (scalar plus vector).

[PRFD \(vector plus immediate\)](#): Gather prefetch doublewords (vector plus immediate).

[PRFH \(scalar plus immediate\)](#): Contiguous prefetch halfwords (immediate index).

[PRFH \(scalar plus scalar\)](#): Contiguous prefetch halfwords (scalar index).

[PRFH \(scalar plus vector\)](#): Gather prefetch halfwords (scalar plus vector).

[PRFH \(vector plus immediate\)](#): Gather prefetch halfwords (vector plus immediate).

[PRFW \(scalar plus immediate\)](#): Contiguous prefetch words (immediate index).

[PRFW \(scalar plus scalar\)](#): Contiguous prefetch words (scalar index).

[PRFW \(scalar plus vector\)](#): Gather prefetch words (scalar plus vector).

[PRFW \(vector plus immediate\)](#): Gather prefetch words (vector plus immediate).

[PSEL](#): Predicate select between predicate register or all-false.

[PTEST](#): Set condition flags for predicate.

[PTRUE \(predicate as counter\)](#): Initialise predicate-as-counter to all active.

[PTRUE \(predicate\)](#): Initialise predicate from named constraint.

[PTRUES](#): Initialise predicate from named constraint and set the condition flags.

[PUNPKHL, PUNPKLO](#): Unpack and widen half of predicate.

[RADDHNB](#): Rounding add narrow high part (bottom).

[RADDHNT](#): Rounding add narrow high part (top).

[RAX1](#): Bitwise rotate left by 1 and exclusive OR.

[RBIT](#): Reverse bits (predicated).

[RDFFR \(predicated\)](#): Return predicate of succesfully loaded elements.

[RDFFR \(unpredicated\)](#): Read the first-fault register.

[RDFFRS](#): Return predicate of succesfully loaded elements, setting the condition flags.

[RDVL](#): Read multiple of vector register size to scalar register.

[REV \(predicate\)](#): Reverse all elements in a predicate.

[REV \(vector\)](#): Reverse all elements in a vector (unpredicated).

[REVB, REVH, REVW](#): Reverse bytes / halfwords / words within elements (predicated).

[REVD](#): Reverse 64-bit doublewords in elements (predicated).

[RSHRNB](#): Rounding shift right narrow by immediate (bottom).

[RSHRNT](#): Rounding shift right narrow by immediate (top).

[RSUBHNB](#): Rounding subtract narrow high part (bottom).

[RSUBHNT](#): Rounding subtract narrow high part (top).

[SABA](#): Signed absolute difference and accumulate.

[SABALB](#): Signed absolute difference and accumulate long (bottom).

[SABALT](#): Signed absolute difference and accumulate long (top).

[SABD](#): Signed absolute difference (predicated).

[SABDLB](#): Signed absolute difference long (bottom).

[SABDLT](#): Signed absolute difference long (top).

[SADALP](#): Signed add and accumulate long pairwise.

[SADDLB](#): Signed add long (bottom).

[SADDLBT](#): Signed add long (bottom + top).

[SADDLT](#): Signed add long (top).

[SADDV](#): Signed add reduction to scalar.

[SADDWB](#): Signed add wide (bottom).

[SADDWT](#): Signed add wide (top).

[SBCLB](#): Subtract with carry long (bottom).

[SBCLT](#): Subtract with carry long (top).

[SCLAMP](#): Signed clamp to minimum/maximum vector.

[SCVTF](#): Signed integer convert to floating-point (predicated).

[SDIV](#): Signed divide (predicated).

[SDIVR](#): Signed reversed divide (predicated).

[SDOT \(2-way, indexed\)](#): Signed integer indexed dot product.

[SDOT \(2-way, vectors\)](#): Signed integer dot product.

[SDOT \(4-way, indexed\)](#): Signed integer indexed dot product.



[SDOT \(4-way, vectors\)](#): Signed integer dot product.

[SEL \(predicates\)](#): Conditionally select elements from two predicates.

[SEL \(vectors\)](#): Conditionally select elements from two vectors.

[SETFFR](#): Initialise the first-fault register to all true.

[SHADD](#): Signed halving addition.

[SHRNB](#): Shift right narrow by immediate (bottom).

[SHRNT](#): Shift right narrow by immediate (top).

[SHSUB](#): Signed halving subtract.

[SHSUBR](#): Signed halving subtract reversed vectors.

[SLI](#): Shift left and insert (immediate).

[SM4E](#): SM4 encryption and decryption.

[SM4EKEY](#): SM4 key updates.

[SMAx \(immediate\)](#): Signed maximum with immediate (unpredicated).

[SMAx \(vectors\)](#): Signed maximum vectors (predicated).

[SMAxP](#): Signed maximum pairwise.

[SMAxQV](#): Signed maximum reduction of quadword vector segments.

[SMAxV](#): Signed maximum reduction to scalar.

[SMIN \(immediate\)](#): Signed minimum with immediate (unpredicated).

[SMIN \(vectors\)](#): Signed minimum vectors (predicated).

[SMINP](#): Signed minimum pairwise.

[SMINQV](#): Signed minimum reduction of quadword vector segments.

[SMINV](#): Signed minimum reduction to scalar.

[SMLALB \(indexed\)](#): Signed multiply-add long to accumulator (bottom, indexed).

[SMLALB \(vectors\)](#): Signed multiply-add long to accumulator (bottom).

[SMLALT \(indexed\)](#): Signed multiply-add long to accumulator (top, indexed).

[SMLALT \(vectors\)](#): Signed multiply-add long to accumulator (top).

[SMLSLB \(indexed\)](#): Signed multiply-subtract long from accumulator (bottom, indexed).

[SMLSLB \(vectors\)](#): Signed multiply-subtract long from accumulator (bottom).

[SMLSLT \(indexed\)](#): Signed multiply-subtract long from accumulator (top, indexed).

[SMLSLT \(vectors\)](#): Signed multiply-subtract long from accumulator (top).

[SMMLA](#): Signed integer matrix multiply-accumulate.

[SMULH \(predicated\)](#): Signed multiply returning high half (predicated).

[SMULH \(unpredicated\)](#): Signed multiply returning high half (unpredicated).

[SMULLB \(indexed\)](#): Signed multiply long (bottom, indexed).

[SMULLB \(vectors\)](#): Signed multiply long (bottom).

[SMULLT \(indexed\)](#): Signed multiply long (top, indexed).



[SMULLT \(vectors\)](#): Signed multiply long (top).

[SPLICE](#): Splice two vectors under predicate control.

[SQABS](#): Signed saturating absolute value.

[SQADD \(immediate\)](#): Signed saturating add immediate (unpredicated).

[SQADD \(vectors, predicated\)](#): Signed saturating addition (predicated).

[SQADD \(vectors, unpredicated\)](#): Signed saturating add vectors (unpredicated).

[SQCADD](#): Saturating complex integer add with rotate.

[SQCVTN](#): Signed saturating extract narrow and interleave.

[SQCVTUN](#): Signed saturating unsigned extract narrow and interleave.

[SQDECB](#): Signed saturating decrement scalar by multiple of 8-bit predicate constraint element count.

[SQDECD \(scalar\)](#): Signed saturating decrement scalar by multiple of 64-bit predicate constraint element count.

[SQDECD \(vector\)](#): Signed saturating decrement vector by multiple of 64-bit predicate constraint element count.

[SQDECH \(scalar\)](#): Signed saturating decrement scalar by multiple of 16-bit predicate constraint element count.

[SQDECH \(vector\)](#): Signed saturating decrement vector by multiple of 16-bit predicate constraint element count.

[SQDECP \(scalar\)](#): Signed saturating decrement scalar by count of true predicate elements.

[SQDECP \(vector\)](#): Signed saturating decrement vector by count of true predicate elements.

[SQDECW \(scalar\)](#): Signed saturating decrement scalar by multiple of 32-bit predicate constraint element count.

[SQDECW \(vector\)](#): Signed saturating decrement vector by multiple of 32-bit predicate constraint element count.

[SQDMLALB \(indexed\)](#): Signed saturating doubling multiply-add long to accumulator (bottom, indexed).

[SQDMLALB \(vectors\)](#): Signed saturating doubling multiply-add long to accumulator (bottom).

[SQDMLALBT](#): Signed saturating doubling multiply-add long to accumulator (bottom × top).

[SQDMLALT \(indexed\)](#): Signed saturating doubling multiply-add long to accumulator (top, indexed).

[SQDMLALT \(vectors\)](#): Signed saturating doubling multiply-add long to accumulator (top).

[SQDMLSLB \(indexed\)](#): Signed saturating doubling multiply-subtract long from accumulator (bottom, indexed).

[SQDMLSLB \(vectors\)](#): Signed saturating doubling multiply-subtract long from accumulator (bottom).

[SQDMLSLBT](#): Signed saturating doubling multiply-subtract long from accumulator (bottom × top).

[SQDMLSLT \(indexed\)](#): Signed saturating doubling multiply-subtract long from accumulator (top, indexed).

[SQDMLSLT \(vectors\)](#): Signed saturating doubling multiply-subtract long from accumulator (top).

[SQDMULH \(indexed\)](#): Signed saturating doubling multiply high (indexed).

[SQDMULH \(vectors\)](#): Signed saturating doubling multiply high (unpredicated).

[SQDMULLB \(indexed\)](#): Signed saturating doubling multiply long (bottom, indexed).

[SQDMULLB \(vectors\)](#): Signed saturating doubling multiply long (bottom).

[SQDMULLT \(indexed\)](#): Signed saturating doubling multiply long (top, indexed).

[SQDMULLT \(vectors\)](#): Signed saturating doubling multiply long (top).

[SQINCB](#): Signed saturating increment scalar by multiple of 8-bit predicate constraint element count.

[SQINCD \(scalar\)](#): Signed saturating increment scalar by multiple of 64-bit predicate constraint element count.

[SQINCD \(vector\)](#): Signed saturating increment vector by multiple of 64-bit predicate constraint element count.

[SQINCH \(scalar\)](#): Signed saturating increment scalar by multiple of 16-bit predicate constraint element count.

[SQINCH \(vector\)](#): Signed saturating increment vector by multiple of 16-bit predicate constraint element count.

[SQINCP \(scalar\)](#): Signed saturating increment scalar by count of true predicate elements.

[SQINCP \(vector\)](#): Signed saturating increment vector by count of true predicate elements.

[SQINCW \(scalar\)](#): Signed saturating increment scalar by multiple of 32-bit predicate constraint element count.

[SQINCW \(vector\)](#): Signed saturating increment vector by multiple of 32-bit predicate constraint element count.

[SQNEG](#): Signed saturating negate.

[SQRDCMLAH \(indexed\)](#): Saturating rounding doubling complex integer multiply-add high with rotate (indexed).

[SQRDCMLAH \(vectors\)](#): Saturating rounding doubling complex integer multiply-add high with rotate.

[SQRDMLAH \(indexed\)](#): Signed saturating rounding doubling multiply-add high to accumulator (indexed).

[SQRDMLAH \(vectors\)](#): Signed saturating rounding doubling multiply-add high to accumulator (unpredicated).

[SQRDMLSH \(indexed\)](#): Signed saturating rounding doubling multiply-subtract high from accumulator (indexed).

[SQRDMLSH \(vectors\)](#): Signed saturating rounding doubling multiply-subtract high from accumulator (unpredicated).

[SQRDMULH \(indexed\)](#): Signed saturating rounding doubling multiply high (indexed).

[SQRDMULH \(vectors\)](#): Signed saturating rounding doubling multiply high (unpredicated).

[SQRSHL](#): Signed saturating rounding shift left by vector (predicated).

[SQRSHLR](#): Signed saturating rounding shift left reversed vectors (predicated).

[SQRSHRN](#): Signed saturating rounding shift right narrow by immediate and interleave.

[SQRSHRNB](#): Signed saturating rounding shift right narrow by immediate (bottom).

[SQRSHRNT](#): Signed saturating rounding shift right narrow by immediate (top).

[SQRSHRUN](#): Signed saturating rounding shift right unsigned narrow by immediate and interleave.

[SQRSHRUNB](#): Signed saturating rounding shift right unsigned narrow by immediate (bottom).

[SQRSHRUNT](#): Signed saturating rounding shift right unsigned narrow by immediate (top).

[SQSHL \(immediate\)](#): Signed saturating shift left by immediate.

[SQSHL \(vectors\)](#): Signed saturating shift left by vector (predicated).

[SQSHLR](#): Signed saturating shift left reversed vectors (predicated).

[SQSHLU](#): Signed saturating shift left unsigned by immediate.

[SQSHRNB](#): Signed saturating shift right narrow by immediate (bottom).

[SQSHRNT](#): Signed saturating shift right narrow by immediate (top).

[SQSHRUNB](#): Signed saturating shift right unsigned narrow by immediate (bottom).

[SQSHRUNT](#): Signed saturating shift right unsigned narrow by immediate (top).

[SQSUB \(immediate\)](#): Signed saturating subtract immediate (unpredicated).

[SQSUB \(vectors, predicated\)](#): Signed saturating subtraction (predicated).

[SQSUB \(vectors, unpredicated\)](#): Signed saturating subtract vectors (unpredicated).

[SQSUBR](#): Signed saturating subtraction reversed vectors (predicated).

[SQXTNB](#): Signed saturating extract narrow (bottom).

[SQXTNT](#): Signed saturating extract narrow (top).

[SQXTUNB](#): Signed saturating unsigned extract narrow (bottom).

[SQXTUNT](#): Signed saturating unsigned extract narrow (top).

[SRHADD](#): Signed rounding halving addition.

[SRI](#): Shift right and insert (immediate).

[SRSHL](#): Signed rounding shift left by vector (predicated).

[SRSHLR](#): Signed rounding shift left reversed vectors (predicated).

[SRSHR](#): Signed rounding shift right by immediate.

[SRSRA](#): Signed rounding shift right and accumulate (immediate).

[SSHLLB](#): Signed shift left long by immediate (bottom).

[SSHLLT](#): Signed shift left long by immediate (top).

[SSRA](#): Signed shift right and accumulate (immediate).

[SSUBLB](#): Signed subtract long (bottom).

[SSUBLBT](#): Signed subtract long (bottom - top).

[SSUBLT](#): Signed subtract long (top).

[SSUBLTB](#): Signed subtract long (top - bottom).

[SSUBWB](#): Signed subtract wide (bottom).

[SSUBWT](#): Signed subtract wide (top).

[ST1B \(scalar plus immediate, consecutive registers\)](#): Contiguous store of bytes from multiple consecutive vectors (immediate index).

[ST1B \(scalar plus immediate, single register\)](#): Contiguous store bytes from vector (immediate index).

[ST1B \(scalar plus scalar, consecutive registers\)](#): Contiguous store of bytes from multiple consecutive vectors (scalar index).

[ST1B \(scalar plus scalar, single register\)](#): Contiguous store bytes from vector (scalar index).

[ST1B \(scalar plus vector\)](#): Scatter store bytes from a vector (vector index).

[ST1B \(vector plus immediate\)](#): Scatter store bytes from a vector (immediate index).

[ST1D \(scalar plus immediate, consecutive registers\)](#): Contiguous store of doublewords from multiple consecutive vectors (immediate index).

[ST1D \(scalar plus immediate, single register\)](#): Contiguous store doublewords from vector (immediate index).

[ST1D \(scalar plus scalar, consecutive registers\)](#): Contiguous store of doublewords from multiple consecutive vectors (scalar index).

[ST1D \(scalar plus scalar, single register\)](#): Contiguous store doublewords from vector (scalar index).

[ST1D \(scalar plus vector\)](#): Scatter store doublewords from a vector (vector index).

[ST1D \(vector plus immediate\)](#): Scatter store doublewords from a vector (immediate index).

[ST1H \(scalar plus immediate, consecutive registers\)](#): Contiguous store of halfwords from multiple consecutive vectors (immediate index).

[ST1H \(scalar plus immediate, single register\)](#): Contiguous store halfwords from vector (immediate index).

[ST1H \(scalar plus scalar, consecutive registers\)](#): Contiguous store of halfwords from multiple consecutive vectors (scalar index).

[ST1H \(scalar plus scalar, single register\)](#): Contiguous store halfwords from vector (scalar index).

[ST1H \(scalar plus vector\)](#): Scatter store halfwords from a vector (vector index).

[ST1H \(vector plus immediate\)](#): Scatter store halfwords from a vector (immediate index).

[ST1Q](#): Scatter store quadwords.

[ST1W \(scalar plus immediate, consecutive registers\)](#): Contiguous store of words from multiple consecutive vectors (immediate index).

[ST1W \(scalar plus immediate, single register\)](#): Contiguous store words from vector (immediate index).

[ST1W \(scalar plus scalar, consecutive registers\)](#): Contiguous store of words from multiple consecutive vectors (scalar index).

[ST1W \(scalar plus scalar, single register\)](#): Contiguous store words from vector (scalar index).

[ST1W \(scalar plus vector\)](#): Scatter store words from a vector (vector index).

[ST1W \(vector plus immediate\)](#): Scatter store words from a vector (immediate index).

[ST2B \(scalar plus immediate\)](#): Contiguous store two-byte structures from two vectors (immediate index).

[ST2B \(scalar plus scalar\)](#): Contiguous store two-byte structures from two vectors (scalar index).

[ST2D \(scalar plus immediate\)](#): Contiguous store two-doubleword structures from two vectors (immediate index).

[ST2D \(scalar plus scalar\)](#): Contiguous store two-doubleword structures from two vectors (scalar index).

[ST2H \(scalar plus immediate\)](#): Contiguous store two-halfword structures from two vectors (immediate index).

[ST2H \(scalar plus scalar\)](#): Contiguous store two-halfword structures from two vectors (scalar index).

[ST2Q \(scalar plus immediate\)](#): Contiguous store two-quadword structures from two vectors (immediate index).

[ST2Q \(scalar plus scalar\)](#): Contiguous store two-quadword structures from two vectors (scalar index).

[ST2W \(scalar plus immediate\)](#): Contiguous store two-word structures from two vectors (immediate index).

[ST2W \(scalar plus scalar\)](#): Contiguous store two-word structures from two vectors (scalar index).

[ST3B \(scalar plus immediate\)](#): Contiguous store three-byte structures from three vectors (immediate index).

[ST3B \(scalar plus scalar\)](#): Contiguous store three-byte structures from three vectors (scalar index).

[ST3D \(scalar plus immediate\)](#): Contiguous store three-doubleword structures from three vectors (immediate index).

[ST3D \(scalar plus scalar\)](#): Contiguous store three-doubleword structures from three vectors (scalar index).

[ST3H \(scalar plus immediate\)](#): Contiguous store three-halfword structures from three vectors (immediate index).

[ST3H \(scalar plus scalar\)](#): Contiguous store three-halfword structures from three vectors (scalar index).

[ST3Q \(scalar plus immediate\)](#): Contiguous store three-quadword structures from three vectors (immediate index).

[ST3Q \(scalar plus scalar\)](#): Contiguous store three-quadword structures from three vectors (scalar index).

[ST3W \(scalar plus immediate\)](#): Contiguous store three-word structures from three vectors (immediate index).

[ST3W \(scalar plus scalar\)](#): Contiguous store three-word structures from three vectors (scalar index).

[ST4B \(scalar plus immediate\)](#): Contiguous store four-byte structures from four vectors (immediate index).

[ST4B \(scalar plus scalar\)](#): Contiguous store four-byte structures from four vectors (scalar index).

[ST4D \(scalar plus immediate\)](#): Contiguous store four-doubleword structures from four vectors (immediate index).

[ST4D \(scalar plus scalar\)](#): Contiguous store four-doubleword structures from four vectors (scalar index).

[ST4H \(scalar plus immediate\)](#): Contiguous store four-halfword structures from four vectors (immediate index).

[ST4H \(scalar plus scalar\)](#): Contiguous store four-halfword structures from four vectors (scalar index).

[ST4Q \(scalar plus immediate\)](#): Contiguous store four-quadword structures from four vectors (immediate index).

[ST4Q \(scalar plus scalar\)](#): Contiguous store four-quadword structures from four vectors (scalar index).

[ST4W \(scalar plus immediate\)](#): Contiguous store four-word structures from four vectors (immediate index).

[ST4W \(scalar plus scalar\)](#): Contiguous store four-word structures from four vectors (scalar index).

[STNT1B \(scalar plus immediate, consecutive registers\)](#): Contiguous store non-temporal of bytes from multiple consecutive vectors (immediate index).

[STNT1B \(scalar plus immediate, single register\)](#): Contiguous store non-temporal bytes from vector (immediate index).

[STNT1B \(scalar plus scalar, consecutive registers\)](#): Contiguous store non-temporal of bytes from multiple consecutive vectors (scalar index).

[STNT1B \(scalar plus scalar, single register\)](#): Contiguous store non-temporal bytes from vector (scalar index).

[STNT1B \(vector plus scalar\)](#): Scatter store non-temporal bytes.

[STNT1D \(scalar plus immediate, consecutive registers\)](#): Contiguous store non-temporal of doublewords from multiple consecutive vectors (immediate index).

[STNT1D \(scalar plus immediate, single register\)](#): Contiguous store non-temporal doublewords from vector (immediate index).

[STNT1D \(scalar plus scalar, consecutive registers\)](#): Contiguous store non-temporal of doublewords from multiple consecutive vectors (scalar index).

[STNT1D \(scalar plus scalar, single register\)](#): Contiguous store non-temporal doublewords from vector (scalar index).

[STNT1D \(vector plus scalar\)](#): Scatter store non-temporal doublewords.

[STNT1H \(scalar plus immediate, consecutive registers\)](#): Contiguous store non-temporal of halfwords from multiple consecutive vectors (immediate index).

[STNT1H \(scalar plus immediate, single register\)](#): Contiguous store non-temporal halfwords from vector (immediate index).

[STNT1H \(scalar plus scalar, consecutive registers\)](#): Contiguous store non-temporal of halfwords from multiple consecutive vectors (scalar index).

[STNT1H \(scalar plus scalar, single register\)](#): Contiguous store non-temporal halfwords from vector (scalar index).

[STNT1H \(vector plus scalar\)](#): Scatter store non-temporal halfwords.

[STNT1W \(scalar plus immediate, consecutive registers\)](#): Contiguous store non-temporal of words from multiple consecutive vectors (immediate index).

[STNT1W \(scalar plus immediate, single register\)](#): Contiguous store non-temporal words from vector (immediate index).

[STNT1W \(scalar plus scalar, consecutive registers\)](#): Contiguous store non-temporal of words from multiple consecutive vectors (scalar index).

[STNT1W \(scalar plus scalar, single register\)](#): Contiguous store non-temporal words from vector (scalar index).

[STNT1W \(vector plus scalar\)](#): Scatter store non-temporal words.

[STR \(predicate\)](#): Store predicate register.

[STR \(vector\)](#): Store vector register.

[SUB \(immediate\)](#): Subtract immediate (unpredicated).

[SUB \(vectors, predicated\)](#): Subtract vectors (predicated).

[SUB \(vectors, unpredicated\)](#): Subtract vectors (unpredicated).

[SUBHNB](#): Subtract narrow high part (bottom).

[SUBHNT](#): Subtract narrow high part (top).

[SUBR \(immediate\)](#): Reversed subtract from immediate (unpredicated).

[SUBR \(vectors\)](#): Reversed subtract vectors (predicated).

[SUDOT](#): Signed by unsigned integer indexed dot product.

[SUNPKHL](#), [SUNPKLO](#): Signed unpack and extend half of vector.

[SUQADD](#): Signed saturating addition of unsigned value.

[SXTB](#), [SXTH](#), [SXTW](#): Signed byte / halfword / word extend (predicated).

[TBL](#): Programmable table lookup in one or two vector table (zeroing).

[TBLQ](#): Programmable table lookup within each quadword vector segment (zeroing).

[TBX](#): Programmable table lookup in single vector table (merging).

[TBXQ](#): Programmable table lookup within each quadword vector segment (merging).

[TRN1](#), [TRN2 \(predicates\)](#): Interleave even or odd elements from two predicates.

[TRN1](#), [TRN2 \(vectors\)](#): Interleave even or odd elements from two vectors.

[UABA](#): Unsigned absolute difference and accumulate.

[UABALB](#): Unsigned absolute difference and accumulate long (bottom).

[UABALT](#): Unsigned absolute difference and accumulate long (top).

[UABD](#): Unsigned absolute difference (predicated).

[UABDLB](#): Unsigned absolute difference long (bottom).

[UABDLT](#): Unsigned absolute difference long (top).

[UADALP](#): Unsigned add and accumulate long pairwise.

[UADDLB](#): Unsigned add long (bottom).

[UADDLT](#): Unsigned add long (top).

[UADDV](#): Unsigned add reduction to scalar.

[UADDWB](#): Unsigned add wide (bottom).

[UADDWT](#): Unsigned add wide (top).

[UCLAMP](#): Unsigned clamp to minimum/maximum vector.

[UCVTF](#): Unsigned integer convert to floating-point (predicated).

[UDIV](#): Unsigned divide (predicated).

[UDIVR](#): Unsigned reversed divide (predicated).

[UDOT \(2-way, indexed\)](#): Unsigned integer indexed dot product.

[UDOT \(2-way, vectors\)](#): Unsigned integer dot product.

[UDOT \(4-way, indexed\)](#): Unsigned integer indexed dot product.

[UDOT \(4-way, vectors\)](#): Unsigned integer dot product.

[UHADD](#): Unsigned halving addition.

[UHSUB](#): Unsigned halving subtract.

[UHSUBR](#): Unsigned halving subtract reversed vectors.

[UMAX \(immediate\)](#): Unsigned maximum with immediate (unpredicated).

[UMAX \(vectors\)](#): Unsigned maximum vectors (predicated).

[UMAXP](#): Unsigned maximum pairwise.

[UMAXQV](#): Unsigned maximum reduction of quadword vector segments.

[UMAXV](#): Unsigned maximum reduction to scalar.

[UMIN \(immediate\)](#): Unsigned minimum with immediate (unpredicated).

[UMIN \(vectors\)](#): Unsigned minimum vectors (predicated).

[UMINP](#): Unsigned minimum pairwise.

[UMINQV](#): Unsigned minimum reduction of quadword vector segments.

[UMINV](#): Unsigned minimum reduction to scalar.

[UMLALB \(indexed\)](#): Unsigned multiply-add long to accumulator (bottom, indexed).

[UMLALB \(vectors\)](#): Unsigned multiply-add long to accumulator (bottom).

[UMLALT \(indexed\)](#): Unsigned multiply-add long to accumulator (top, indexed).

[UMLALT \(vectors\)](#): Unsigned multiply-add long to accumulator (top).

[UMLSLB \(indexed\)](#): Unsigned multiply-subtract long from accumulator (bottom, indexed).

[UMLSLB \(vectors\)](#): Unsigned multiply-subtract long from accumulator (bottom).

[UMLSLT \(indexed\)](#): Unsigned multiply-subtract long from accumulator (top, indexed).

[UMLSLT \(vectors\)](#): Unsigned multiply-subtract long from accumulator (top).

[UMMLA](#): Unsigned integer matrix multiply-accumulate.

[UMULH \(predicated\)](#): Unsigned multiply returning high half (predicated).

[UMULH \(unpredicated\)](#): Unsigned multiply returning high half (unpredicated).

[UMULLB \(indexed\)](#): Unsigned multiply long (bottom, indexed).

[UMULLB \(vectors\)](#): Unsigned multiply long (bottom).

[UMULLT \(indexed\)](#): Unsigned multiply long (top, indexed).

[UMULLT \(vectors\)](#): Unsigned multiply long (top).

[UQADD \(immediate\)](#): Unsigned saturating add immediate (unpredicated).

[UQADD \(vectors, predicated\)](#): Unsigned saturating addition (predicated).

[UQADD \(vectors, unpredicated\)](#): Unsigned saturating add vectors (unpredicated).

[UQCVTN](#): Unsigned saturating extract narrow and interleave.

[UQDECB](#): Unsigned saturating decrement scalar by multiple of 8-bit predicate constraint element count.

[UQDECD \(scalar\)](#): Unsigned saturating decrement scalar by multiple of 64-bit predicate constraint element count.

[UQDECD \(vector\)](#): Unsigned saturating decrement vector by multiple of 64-bit predicate constraint element count.

[UQDECH \(scalar\)](#): Unsigned saturating decrement scalar by multiple of 16-bit predicate constraint element count.

[UQDECH \(vector\)](#): Unsigned saturating decrement vector by multiple of 16-bit predicate constraint element count.

[UQDECP \(scalar\)](#): Unsigned saturating decrement scalar by count of true predicate elements.



[UQDECP \(vector\)](#): Unsigned saturating decrement vector by count of true predicate elements.

[UQDECW \(scalar\)](#): Unsigned saturating decrement scalar by multiple of 32-bit predicate constraint element count.

[UQDECW \(vector\)](#): Unsigned saturating decrement vector by multiple of 32-bit predicate constraint element count.

[UQINCB](#): Unsigned saturating increment scalar by multiple of 8-bit predicate constraint element count.

[UQINCD \(scalar\)](#): Unsigned saturating increment scalar by multiple of 64-bit predicate constraint element count.

[UQINCD \(vector\)](#): Unsigned saturating increment vector by multiple of 64-bit predicate constraint element count.

[UQINCH \(scalar\)](#): Unsigned saturating increment scalar by multiple of 16-bit predicate constraint element count.

[UQINCH \(vector\)](#): Unsigned saturating increment vector by multiple of 16-bit predicate constraint element count.

[UQINCP \(scalar\)](#): Unsigned saturating increment scalar by count of true predicate elements.

[UQINCP \(vector\)](#): Unsigned saturating increment vector by count of true predicate elements.

[UQINCW \(scalar\)](#): Unsigned saturating increment scalar by multiple of 32-bit predicate constraint element count.

[UQINCW \(vector\)](#): Unsigned saturating increment vector by multiple of 32-bit predicate constraint element count.

[UQRSHL](#): Unsigned saturating rounding shift left by vector (predicated).

[UQRSHLR](#): Unsigned saturating rounding shift left reversed vectors (predicated).

[UQRSHRN](#): Unsigned saturating rounding shift right narrow by immediate and interleave.

[UQRSHRNB](#): Unsigned saturating rounding shift right narrow by immediate (bottom).

[UQRSHRNT](#): Unsigned saturating rounding shift right narrow by immediate (top).

[UQSHL \(immediate\)](#): Unsigned saturating shift left by immediate.

[UQSHL \(vectors\)](#): Unsigned saturating shift left by vector (predicated).

[UQSHLR](#): Unsigned saturating shift left reversed vectors (predicated).

[UQSHRNB](#): Unsigned saturating shift right narrow by immediate (bottom).

[UQSHRNT](#): Unsigned saturating shift right narrow by immediate (top).

[UQSUB \(immediate\)](#): Unsigned saturating subtract immediate (unpredicated).

[UQSUB \(vectors, predicated\)](#): Unsigned saturating subtraction (predicated).

[UQSUB \(vectors, unpredicated\)](#): Unsigned saturating subtract vectors (unpredicated).

[UQSUBR](#): Unsigned saturating subtraction reversed vectors (predicated).

[UQXTNB](#): Unsigned saturating extract narrow (bottom).

[UQXTNT](#): Unsigned saturating extract narrow (top).

[URECPE](#): Unsigned reciprocal estimate (predicated).

[URHADD](#): Unsigned rounding halving addition.

[URSHL](#): Unsigned rounding shift left by vector (predicated).

[URSHLR](#): Unsigned rounding shift left reversed vectors (predicated).

[URSHR](#): Unsigned rounding shift right by immediate.

[URSQRTE](#): Unsigned reciprocal square root estimate (predicated).

[URSRA](#): Unsigned rounding shift right and accumulate (immediate).

[USDOT \(indexed\)](#): Unsigned by signed integer indexed dot product.



[USDOT \(vectors\)](#): Unsigned by signed integer dot product.

[USHLLB](#): Unsigned shift left long by immediate (bottom).

[USHLLT](#): Unsigned shift left long by immediate (top).

[USMMLA](#): Unsigned by signed integer matrix multiply-accumulate.

[USQADD](#): Unsigned saturating addition of signed value.

[USRA](#): Unsigned shift right and accumulate (immediate).

[USUBLB](#): Unsigned subtract long (bottom).

[USUBLT](#): Unsigned subtract long (top).

[USUBWB](#): Unsigned subtract wide (bottom).

[USUBWT](#): Unsigned subtract wide (top).

[UUNPKHI](#), [UUNPKLO](#): Unsigned unpack and extend half of vector.

[UXTB](#), [UXTH](#), [UXTW](#): Unsigned byte / halfword / word extend (predicated).

[UZP1](#), [UZP2 \(predicates\)](#): Concatenate even or odd elements from two predicates.

[UZP1](#), [UZP2 \(vectors\)](#): Concatenate even or odd elements from two vectors.

[UZPO1](#): Concatenate even elements within each pair of quadword vector segments.

[UZPO2](#): Concatenate odd elements within each pair of quadword vector segments.

[WHILEGE \(predicate as counter\)](#): While decrementing signed scalar greater than or equal to scalar (predicate-as-counter).

[WHILEGE \(predicate pair\)](#): While decrementing signed scalar greater than or equal to scalar (pair of predicates).

[WHILEGE \(predicate\)](#): While decrementing signed scalar greater than or equal to scalar.

[WHILEGT \(predicate as counter\)](#): While decrementing signed scalar greater than scalar (predicate-as-counter).

[WHILEGT \(predicate pair\)](#): While decrementing signed scalar greater than scalar (pair of predicates).

[WHILEGT \(predicate\)](#): While decrementing signed scalar greater than scalar.

[WHILEHI \(predicate as counter\)](#): While decrementing unsigned scalar higher than scalar (predicate-as-counter).

[WHILEHI \(predicate pair\)](#): While decrementing unsigned scalar higher than scalar (pair of predicates).

[WHILEHI \(predicate\)](#): While decrementing unsigned scalar higher than scalar.

[WHILEHS \(predicate as counter\)](#): While decrementing unsigned scalar higher or same as scalar (predicate-as-counter).

[WHILEHS \(predicate pair\)](#): While decrementing unsigned scalar higher or same as scalar (pair of predicates).

[WHILEHS \(predicate\)](#): While decrementing unsigned scalar higher or same as scalar.

[WHILELE \(predicate as counter\)](#): While incrementing signed scalar less than or equal to scalar (predicate-as-counter).

[WHILELE \(predicate pair\)](#): While incrementing signed scalar less than or equal to scalar (pair of predicates).

[WHILELE \(predicate\)](#): While incrementing signed scalar less than or equal to scalar.

[WHILELO \(predicate as counter\)](#): While incrementing unsigned scalar lower than scalar (predicate-as-counter).

[WHILELO \(predicate pair\)](#): While incrementing unsigned scalar lower than scalar (pair of predicates).

[WHILELO \(predicate\)](#): While incrementing unsigned scalar lower than scalar.

[WHILELS \(predicate as counter\)](#): While incrementing unsigned scalar lower or same as scalar (predicate-as-counter).

[WHILELS \(predicate pair\)](#): While incrementing unsigned scalar lower or same as scalar (pair of predicates).

[WHILELS \(predicate\)](#): While incrementing unsigned scalar lower or same as scalar.

[WHILELT \(predicate as counter\)](#): While incrementing signed scalar less than scalar (predicate-as-counter).

[WHILELT \(predicate pair\)](#): While incrementing signed scalar less than scalar (pair of predicates).

[WHILELT \(predicate\)](#): While incrementing signed scalar less than scalar.

[WHILERW](#): While free of read-after-write conflicts.

[WHILEWR](#): While free of write-after-read/write conflicts.

[WRFFR](#): Write the first-fault register.

[XAR](#): Bitwise exclusive OR and rotate right by immediate.

[ZIP1, ZIP2 \(predicates\)](#): Interleave elements from two half predicates.

[ZIP1, ZIP2 \(vectors\)](#): Interleave elements from two half vectors.

[ZIPQ1](#): Interleave elements from low halves of each pair of quadword vector segments.

[ZIPQ2](#): Interleave elements from high halves of each pair of quadword vector segments.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ABS

Absolute value (predicated)

Compute the absolute value of the signed integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	0	1	1	0	1	0	1	Pg													

ABS <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer element = SInt(Elem[operand, e, esize]);
        element = Abs(element);
        Elem[result, e, esize] = element<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

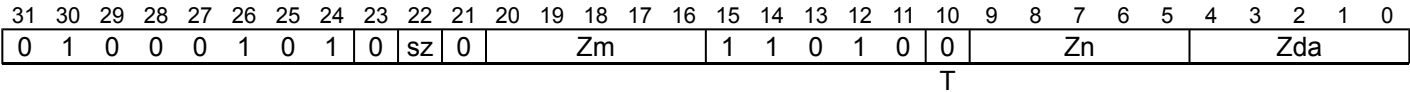
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADCLB

Add with carry long (bottom)

Add the even-numbered elements of the first source vector and the 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector to the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.



```
ADCLB <Zda>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand = Z[n, VL];
bits(VL) carries = Z[m, VL];
bits(VL) result = Z[da, VL];

for p = 0 to pairs-1
    bits(esize) element1 = Elem[result, 2*p + 0, esize];
    bits(esize) element2 = Elem[operand, 2*p + 0, esize];
    bit carry_in = Elem[carries, 2*p + 1, esize]<0>;

    (res, nzcvc) = AddWithCarry(element1, element2, carry_in);
    carry_out = nzcvc<1>;

    Elem[result, 2*p + 0, esize] = res;
    Elem[result, 2*p + 1, esize] = ZeroExtend(carry_out, esize);

Z[da, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

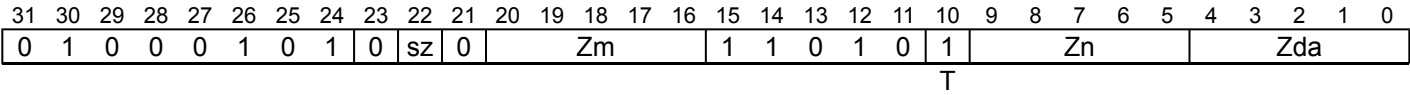
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADCLT

Add with carry long (top)

Add the odd-numbered elements of the first source vector and the 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector to the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.



```
ADCLT <Zda>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand = Z[n, VL];
bits(VL) carries = Z[m, VL];
bits(VL) result = Z[da, VL];

for p = 0 to pairs-1
    bits(esize) element1 = Elem[result, 2*p + 0, esize];
    bits(esize) element2 = Elem[operand, 2*p + 1, esize];
    bit carry_in = Elem[carries, 2*p + 1, esize]<0>;

    (res, nzcvc) = AddWithCarry(element1, element2, carry_in);
    carry_out = nzcvc<1>;

    Elem[result, 2*p + 0, esize] = res;
    Elem[result, 2*p + 1, esize] = ZeroExtend(carry_out, esize);

Z[da, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



ADD (immediate)

Add immediate (unpredicated)

Add an unsigned immediate to each element of the source vector, and destructively place the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	0	0	0	0	0	1	1	sh	imm8								Zdn					

ADD <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    Elem[result, e, esize] = element1 + imm;

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ADD (vectors, predicated)

Add vectors (predicated)

Add active elements of the second source vector to corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	0	0	0	0	0	0	0	0	0	Pg												

ADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

### Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = element1 + element2;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```

### Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD (vectors, unpredicated)

Add vectors (unpredicated)

Add all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1				Zm			0	0	0	0	0	0			Zn					Zd		

```
ADD <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = element1 + element2;

Z[d, VL] = result;
```

Operational information

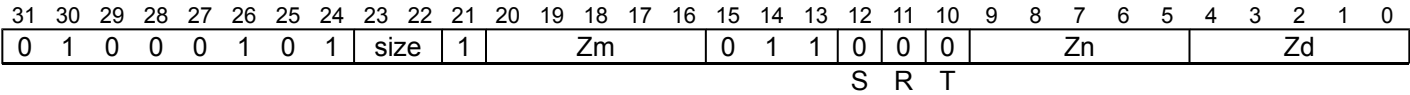
If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

ADDHNB

Add narrow high part (bottom)

Add each vector element of the first source vector to the corresponding vector element of the second source vector, and place the most significant half of the result in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. This instruction is unpredicated.



```
ADDHNB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = (element1 + element2) >> halfesize;
    Elem[result, 2*e + 0, halfesize] = res<halfesize-1:0>;
    Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

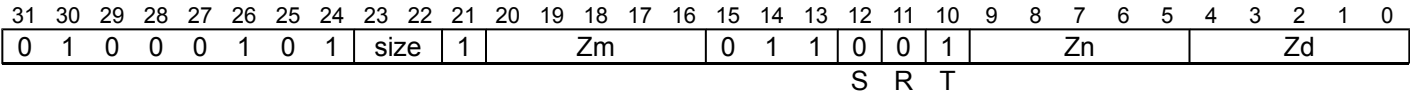
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADDHNT

Add narrow high part (top)

Add each vector element of the first source vector to the corresponding vector element of the second source vector, and place the most significant half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.



```
ADDHNT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = (element1 + element2) >> halfesize;
    Elem[result, 2*e + 1, halfesize] = res<halfesize-1:0>;

Z[d, VL] = result;
```



## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

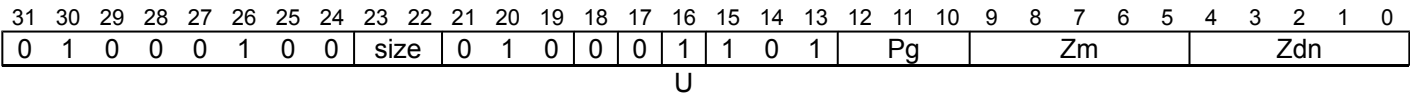
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADDP

Add pairwise

Add pairs of adjacent elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.



```
ADDP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
integer element1;
integer element2;

for e = 0 to elements-1
  if !ActivePredicateElement(mask, e, esize) then
    Elem[result, e, esize] = Elem[operand1, e, esize];
  else
    if IsEven(e) then
      element1 = UInt(Elem[operand1, e + 0, esize]);
      element2 = UInt(Elem[operand1, e + 1, esize]);
    else
      element1 = UInt(Elem[operand2, e - 1, esize]);
      element2 = UInt(Elem[operand2, e + 0, esize]);
    integer res = element1 + element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[dn, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADDPL

Add multiple of predicate register size to scalar register

Add the current predicate register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer and place the result in the 64-bit destination general-purpose register or current stack pointer.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	1	1					Rn		0	1	0	1	0										Rd

```
ADDPL <Xd|SP>, <Xn|SP>, #<imm>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm6);
```

Assembler Symbols

- <Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) result = operand1 + (imm * (PL DIV 8));

if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

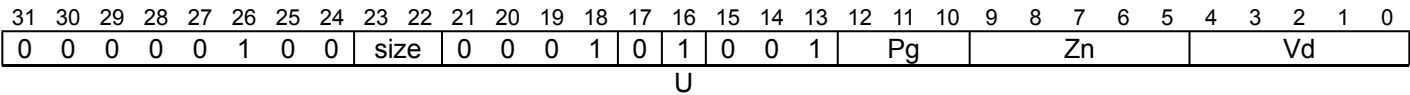
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

ADDQV

Unsigned add reduction of quadword vector segments

Unsigned addition of the same element numbers from each 128-bit source vector segment, placing each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as zero.

SVE2  
(FEAT\_SVE2p1)



```
ADDQV <Vd>.<T>, <Pg>, <Zn>.<Tb>
```

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = TRUE;
```

Assembler Symbols

<Vd> Is the name of the destination SIMD&FP register, encoded in the "Vd" field.

<T> Is an arrangement specifier, encoded in "size":

size	<T>
00	16B
01	8H
10	4S
11	2D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	B
01	H
10	S
11	D

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elemperssegment = 128 DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(128) result = Zeros(128);
bits(128) stmp = Zeros(128);

integer dtmp;

for e = 0 to elemperssegment-1
    dtmp = 0;
    for s = 0 to segments-1
        if ActivePredicateElement(mask, s * elemperssegment + e, esize) then
            stmp = Elem[operand, s, 128];
            dtmp = dtmp + UInt(Elem[stmp, e, esize]);
        Elem[result, e, esize] = dtmp<esize-1:0>;

V[d, 128] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADDVL

Add multiple of vector register size to scalar register

Add the current vector register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer, and place the result in the 64-bit destination general-purpose register or current stack pointer.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	1					Rn		0	1	0	1	0										Rd

```
ADDVL <Xd|SP>, <Xn|SP>, #<imm>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm6);
```

Assembler Symbols

- <Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) result = operand1 + (imm * (VL DIV 8));

if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

## ADR

Compute vector address

Optionally sign or zero-extend the least significant 32-bits of each element from a vector of offsets or indices in the second source vector, scale each index by 2, 4 or 8, add to a vector of base addresses from the first source vector, and place the resulting addresses in the destination vector. This instruction is unpredicated.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 3 classes: [Packed offsets](#) , [Unpacked 32-bit signed offsets](#) and [Unpacked 32-bit unsigned offsets](#)

### Packed offsets

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	sz	1			Zm			1	0	1	0	msz				Zn					Zd		

ADR <Zd>.<T>, [<Zn>.<T>, <Zm>.<T>{, <mod> <amount>}]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
constant integer osize = esize;
boolean unsigned = TRUE;
integer mbytes = 1 << UInt(msz);
```

### Unpacked 32-bit signed offsets

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	1			Zm			1	0	1	0	msz				Zn					Zd		

ADR <Zd>.D, [<Zn>.D, <Zm>.D, SXTW{ <amount>}]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
constant integer osize = 32;
boolean unsigned = FALSE;
integer mbytes = 1 << UInt(msz);
```

### Unpacked 32-bit unsigned offsets

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	1	1			Zm			1	0	1	0	msz				Zn					Zd		

ADR <Zd>.D, [<Zn>.D, <Zm>.D, UXTW{ <amount>}]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
constant integer osize = 32;
boolean unsigned = TRUE;
integer mbytes = 1 << UInt(msz);
```



Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
- <mod> Is the index extend and shift specifier, encoded in "msz":

msz	<mod>
00	[absent]
x1	LSL
10	LSL

- <amount> Is the index shift amount, encoded in "msz":

msz	<amount>
00	[absent]
01	#1
10	#2
11	#3

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) base = Z[n, VL];
bits(VL) offs = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) addr = Elem[base, e, esize];
    integer offset = Int(Elem[offs, e, esize]<osize-1:0>, unsigned);
    Elem[result, e, esize] = addr + (offset * mbytes);

Z[d, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

AESD

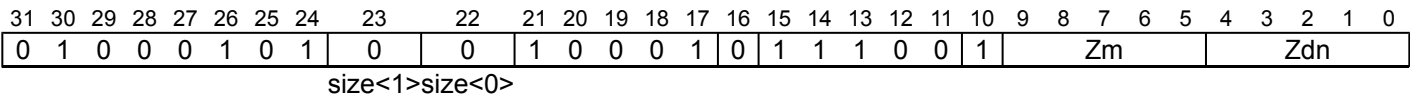
AES single round decryption

The `AESD` instruction reads a 16-byte state array from each 128-bit segment of the first source vector, together with a round key from the corresponding 128-bit segment of the second source vector. Each state array undergoes a single round of the `ADDROUNDKEY()`, `INVSUBBYTES()` and `INVSHIFTRROWS()` transformations in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

`ID_AA64ZFR0_EL1.AES` indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless `FEAT_SME_FA64` is implemented and enabled.

SVE2  
(FEAT\_SVE\_AES)



```
AESD <Zdn>.B, <Zdn>.B, <Zm>.B
```

```
if !HaveSVE() || !HaveSVE2AES() then UNDEFINED;
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

result = operand1 EOR operand2;
for s = 0 to segments-1
    Elem[result, s, 128] = AESInvSubBytes(AESInvShiftRows(Elem[result, s, 128]));

Z[dn, VL] = result;
```

Operational information

- If `PSTATE.DIT` is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

AESE

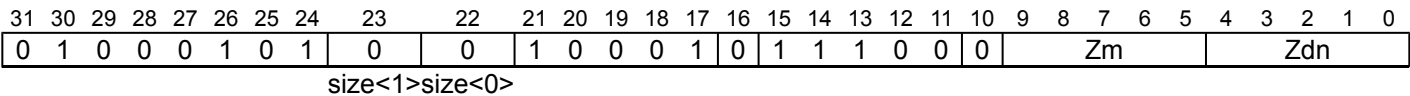
AES single round encryption

The `AESE` instruction reads a 16-byte state array from each 128-bit segment of the first source vector together with a round key from the corresponding 128-bit segment of the second source vector. Each state array undergoes a single round of the `ADDROUNDKEY()`, `SUBBYTES()` and `SHIFTRROWS()` transformations in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

`ID_AA64ZFR0_EL1.AES` indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless `FEAT_SME_FA64` is implemented and enabled.

SVE2  
(FEAT\_SVE\_AES)



```
AESE <Zdn>.B, <Zdn>.B, <Zm>.B
```

```
if !HaveSVE() || !HaveSVE2AES() then UNDEFINED;
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

result = operand1 EOR operand2;
for s = 0 to segments-1
    Elem[result, s, 128] = AESSubBytes(AESShiftRows(Elem[result, s, 128]));

Z[dn, VL] = result;
```

Operational information

- If `PSTATE.DIT` is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

AESIMC

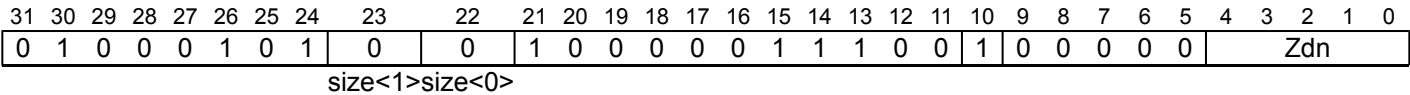
AES inverse mix columns

The AESIMC instruction reads a 16-byte state array from each 128-bit segment of the source register, and performs a single round of the INVMIXCOLUMNS() transformation on each state array in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.AES indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE2
(FEAT\_SVE\_AES)



AESIMC <Zdn>.B, <Zdn>.B

```
if !HaveSVE() || !HaveSVE2AES() then UNDEFINED;
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand = Z[dn, VL];
bits(VL) result;

for s = 0 to segments-1
    Elem[result, s, 128] = AESInvMixColumns(Elem[operand, s, 128]);

Z[dn, VL] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

AESMC

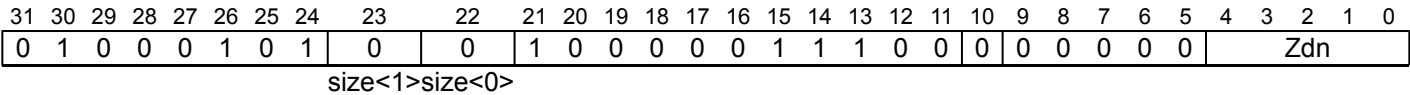
AES mix columns

The AESMC instruction reads a 16-byte state array from each 128-bit segment of the source register, and performs a single round of the MIXCOLUMNS() transformation on each state array in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.AES indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE2  
(FEAT\_SVE\_AES)



AESMC <Zdn>.B, <Zdn>.B

```
if !HaveSVE() || !HaveSVE2AES() then UNDEFINED;
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand = Z[dn, VL];
bits(VL) result;

for s = 0 to segments-1
    Elem[result, s, 128] = AESMixColumns(Elem[operand, s, 128]);

Z[dn, VL] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

AND (immediate)

Bitwise AND with immediate (unpredicated)

Bitwise AND an immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction [BIC \(immediate\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	1	0	0	0	0	0	imm13														Zdn			

AND <Zdn>.<T>, <Zdn>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer dn = UInt(Zdn);
bits(64) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE, 64);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "imm13<12>:imm13<5:0>":

imm13<12>	imm13<5:0>	<T>
0	0xxxxxx	S
0	10xxxxx	H
0	110xxxx	B
0	1110xxx	B
0	11110xx	B
0	1111110	RESERVED
0	1111111	RESERVED
1	xxxxxxx	D

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 64;
bits(VL) operand = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(64) element1 = Elem[operand, e, 64];
    Elem[result, e, 64] = element1 AND imm;

Z[dn, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

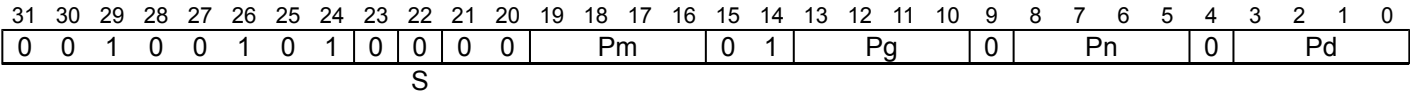
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AND (predicates)

Bitwise AND predicates

Bitwise AND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This instruction is used by the alias [MOV \(predicate, predicated, zeroing\)](#).



```
AND <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
<a href="#">MOV (predicate, predicated, zeroing)</a>	S == '0' && Pn == Pm

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit element1 = PredicateElement(operand1, e, esize);
    bit element2 = PredicateElement(operand2, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, psize] = ZeroExtend(element1 AND element2, psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```



## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AND (vectors, predicated)

Bitwise AND vectors (predicated)

Bitwise AND active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	0	1	0	0	0	0	0	Pg												

AND <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  if ActivePredicateElement(mask, e, esize) then
    Elem[result, e, esize] = element1 AND element2;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AND (vectors, unpredicated)

Bitwise AND vectors (unpredicated)

Bitwise AND all elements of the second source vector with corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	1	Zm				0	0	1	1	0	0	Zn				Zd						

AND <Zd>.D, <Zn>.D, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];

Z[d, VL] = operand1 AND operand2;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# ANDQV

Bitwise AND reduction of quadword vector segments

Bitwise AND of the same element numbers from each 128-bit source vector segment, placing each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as all ones.

## SVE2 (FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	1	1	0	0	0	1	Pg	Zn				Vd								

ANDQV <Vd>.<T>, <Pg>, <Zn>.<Tb>

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

## Assembler Symbols

<Vd> Is the name of the destination SIMD&FP register, encoded in the "Vd" field.

<T> Is an arrangement specifier, encoded in "size":

size	<T>
00	16B
01	8H
10	4S
11	2D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	B
01	H
10	S
11	D

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elemperssegment = 128 DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(128) result = Zeros(128);
bits(128) stmp = Zeros(128);

bits(esize) dtmp;

for e = 0 to elemperssegment-1
    dtmp = Ones(esize);
    for s = 0 to segments-1
        if ActivePredicateElement(mask, s * elemperssegment + e, esize) then
            stmp = Elem[operand, s, 128];
            dtmp = dtmp AND Elem[stmp, e, esize];
        Elem[result, e, esize] = dtmp<esize-1:0>;

V[d, 128] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

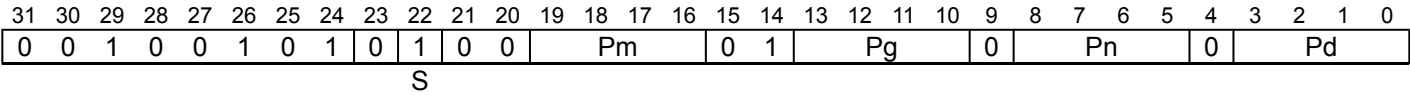
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ANDS

Bitwise AND predicates, setting the condition flags

Bitwise AND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the alias [MOVS \(predicated\)](#).



ANDS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
<a href="#">MOVS (predicated)</a>	S == '1' && Pn == Pm

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit element1 = PredicateElement(operand1, e, esize);
    bit element2 = PredicateElement(operand2, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, psize] = ZeroExtend(element1 AND element2, psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



ANDV

Bitwise AND reduction to scalar

Bitwise AND horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as all ones.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size		0	1	1	0	1	0	0	0	1		Pg											

ANDV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) result = Ones(esize);

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        result = result AND Elem[operand, e, esize];

V[d, esize] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

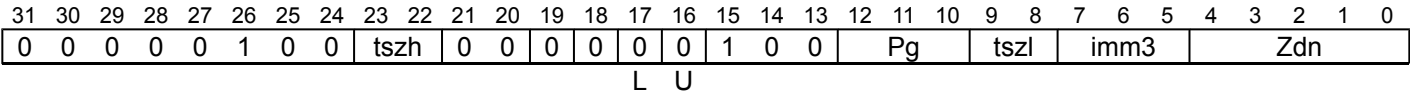
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASR (immediate, predicated)

Arithmetic shift right by immediate (predicated)

Shift right by immediate, preserving the sign bit, each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.



```
ASR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer PL = VL DIV 8;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = ASR(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

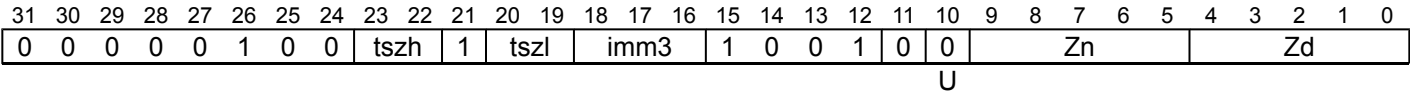
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASR (immediate, unpredicated)

Arithmetic shift right by immediate (unpredicated)

Shift right by immediate, preserving the sign bit, each element of the source vector, and place the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



```
ASR <Zd>.<T>, <Zn>.<T>, #<const>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    Elem[result, e, esize] = ASR(element1, shift);

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

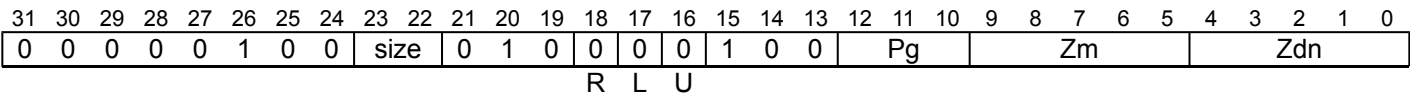
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



ASR (vectors)

Arithmetic shift right by vector (predicated)

Shift right, preserving the sign bit, active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.



```
ASR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        integer shift = Min(UInt(element2), esize);
        Elem[result, e, esize] = ASR(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



ASR (wide elements, predicated)

Arithmetic shift right by 64-bit wide elements (predicated)

Shift right, preserving the sign bit, active elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	0	0	0	1	0	0	Pg			Zm				Zdn						
R																L				U											

ASR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	RESERVED

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
    integer shift = Min(UInt(element2), esize);
    Elem[result, e, esize] = ASR(element1, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and destination element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

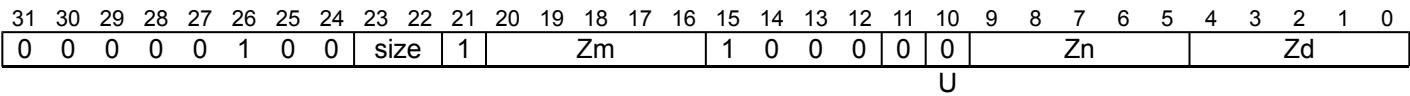
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASR (wide elements, unpredicated)

Arithmetic shift right by 64-bit wide elements (unpredicated)

Shift right, preserving the sign bit, all elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and place the first in the corresponding elements of the destination vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. This instruction is unpredicated.



ASR <Zd>.<T>, <Zn>.<T>, <Zm>.<D>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	RESERVED

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
    integer shift = Min(UInt(element2), esize);
    Elem[result, e, esize] = ASR(element1, shift);

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

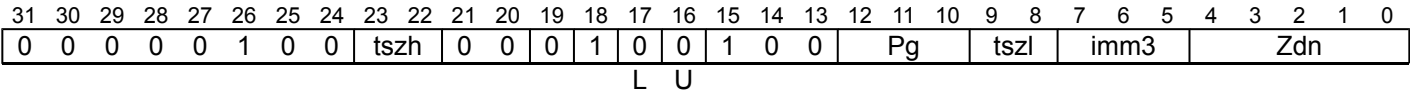
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASRD

Arithmetic shift right for divide by immediate (predicated)

Shift right by immediate, preserving the sign bit, each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The result rounds toward zero as in a signed division. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.



ASRD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    integer element1 = SInt(Elem[operand1, e, esize]);
    if element1 < 0 then
      element1 = element1 + ((1 << shift) - 1);
    Elem[result, e, esize] = (element1 >> shift)<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

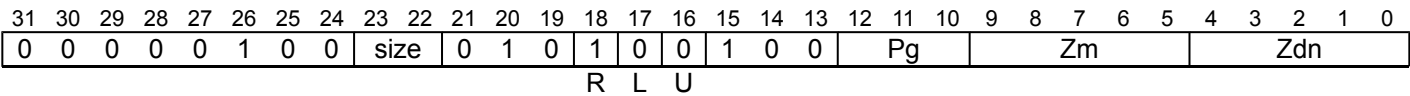
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASRR

Reversed arithmetic shift right by vector (predicated)

Reversed shift right, preserving the sign bit, active elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.



ASRR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        integer shift = Min(UInt(element1), esize);
        Elem[result, e, esize] = ASR(element2, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



BCAX

Bitwise clear and exclusive OR

Bitwise AND elements of the second source vector with the corresponding inverted elements of the third source vector, then exclusive OR the results with corresponding elements of the first source vector. The final results are destructively placed in the corresponding elements of the destination and first source vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	1	1	Zm				0	0	1	1	1	0	Zk				Zdn						

BCAX <Zdn>.D, <Zdn>.D, <Zm>.D, <Zk>.D

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);
```

Assembler Symbols

- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <Zk> Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[k, VL];

Z[dn, VL] = operand1 EOR (operand2 AND NOT(operand3));
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

BDEP

Scatter lower bits into positions selected by bitmask

This instruction scatters the lowest-numbered contiguous bits within each element of the first source vector to the bit positions indicated by non-zero bits in the corresponding mask element of the second source vector, preserving their order, and set the bits corresponding to a zero mask bit to zero. This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.BitPerm indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE2
(FEAT\_SVE\_BitPerm)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 1 size 0 Zm 1 0 1 1 0 1 Zn Zd

BDEP <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() || !HaveSVE2BitPerm() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

Table with 2 columns: size, <T>. Rows: 00 B, 01 H, 10 S, 11 D.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) data = Z[n, VL];
bits(VL) mask = Z[m, VL];
bits(VL) result;

for e = 0 to elements - 1
 Elem[result, e, esize] = BitDeposit(Elem[data, e, esize], Elem[mask, e, esize]);

Z[d, VL] = result;

Operational information

- If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
◦ The values of the data supplied in any of its registers.
◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BEXT

Gather lower bits from positions selected by bitmask

This instruction gathers bits in each element of the first source vector from the bit positions indicated by non-zero bits in the corresponding mask element of the second source vector to the lowest-numbered contiguous bits of the corresponding destination element, preserving their order, and sets the remaining higher-numbered bits to zero. This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.BitPerm indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE2
(FEAT\_SVE\_BitPerm)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 1 size 0 Zm 1 0 1 1 0 0 Zn Zd

BEXT <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() || !HaveSVE2BitPerm() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

Table with 2 columns: size, <T>. Rows: 00 B, 01 H, 10 S, 11 D.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) data = Z[n, VL];
bits(VL) mask = Z[m, VL];
bits(VL) result;

for e = 0 to elements - 1
 Elem[result, e, esize] = BitExtract(Elem[data, e, esize], Elem[mask, e, esize]);

Z[d, VL] = result;

Operational information

- If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
◦ The values of the data supplied in any of its registers.
◦ The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BFADD (predicated)

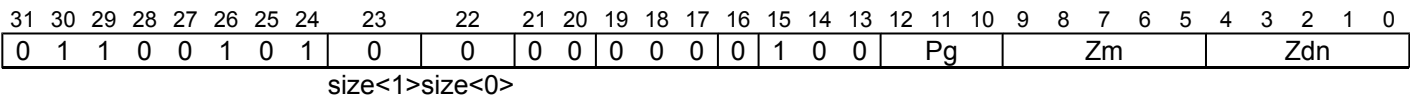
BFloat16 floating-point add vectors (predicated)

Add active BFloat16 elements of the second source vector to corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

This instruction follows SVE2.1 non-widening BFloat16 numerical behaviors.

ID\_AA64ZFR0\_EL1.B16B16 indicates whether this instruction is implemented.

## SVE2 (FEAT\_SVE\_B16B16)



BFADD <Zdn>.H, <Pg>/M, <Zdn>.H, <Zm>.H

```
if (!HaveSVE2() && !HaveSME2()) || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

## Assembler Symbols

- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 16;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, 16) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, e, 16];
    if ActivePredicateElement(mask, e, 16) then
        bits(16) element2 = Elem[operand2, e, 16];
        Elem[result, e, 16] = BFAdd(element1, element2, FPCR[]);
    else
        Elem[result, e, 16] = element1;

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



BFADD (unpredicated)

BFloat16 floating-point add vectors (unpredicated)

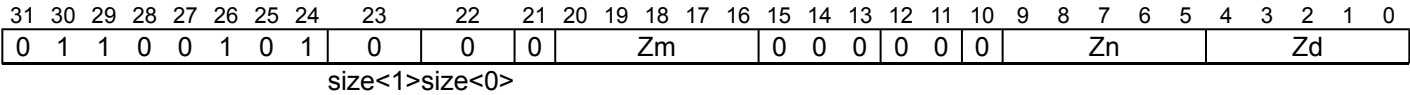
Add all BFloat16 elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector.

This instruction follows SVE2.1 non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.B16B16 indicates whether this instruction is implemented.

SVE2
(FEAT\_SVE\_B16B16)



BFADD <Zd>.H, <Zn>.H, <Zm>.H

```
if (!HaveSVE2() && !HaveSME2()) || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, e, 16];
    bits(16) element2 = Elem[operand2, e, 16];
    Elem[result, e, 16] = BFAdd(element1, element2, FPCR[]);

Z[d, VL] = result;
```



BFCLAMP

BFloat16 floating-point clamp to minimum/maximum number

Clamp each BFloat16 element in the destination vector to between the BFloat16 minimum value in the corresponding element of the first source vector and the BFloat16 maximum value in the corresponding element of the second source vector and destructively place the clamped results in the corresponding elements of the destination vector. Regardless of the value of FPCR.AH, the behavior is as follows for each minimum number and maximum number operation:

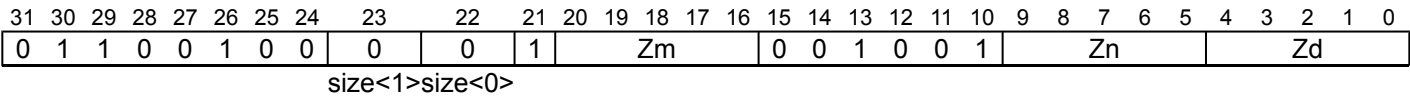
- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction follows SVE2.1 non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.B16B16 indicates whether this instruction is implemented.

SVE2
(FEAT\_SVE\_B16B16)



BFCLAMP <Zd>.H, <Zn>.H, <Zm>.H

```
if (!HaveSVE2() && !HaveSME2()) || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 16;
bits(VL) result;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[d, VL];

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, e, 16];
    bits(16) element2 = Elem[operand2, e, 16];
    bits(16) element3 = Elem[operand3, e, 16];
    Elem[result, e, 16] = BMinNum(BMaxNum(element1, element3, FPCR[]), element2, FPCR[]);
Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFCVT

Floating-point down convert to BFloat16 format (predicated)

Convert to BFloat16 from single-precision in each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

Since the result type is smaller than the input type, the results are zero-extended to fill each destination element. ID\_AA64ZFR0\_EL1.BF16 indicates whether this instruction is implemented.

SVE
(FEAT\_BF16)

Table with 32 columns (31 to 0) and 1 row. Columns 31-24 contain bits 0-7. Columns 23-16 contain bits 8-15. Columns 15-12 contain the Pg field. Columns 11-8 contain the Zn field. Columns 7-0 contain the Zd field.

BFCVT <Zd>.H, <Pg>/M, <Zn>.S

```
if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, 32) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, 32) then
    bits(32) element = Elem[operand, e, 32];
    Elem[result, 2*e, 16] = FPConvertBF(element, FPCR[]);
    Elem[result, 2*e+1, 16] = Zeros(16);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

BFCVTNT

Floating-point down convert and narrow to BFloat16 (top, predicated)

Convert to BFloat16 from single-precision in each active floating-point element of the source vector, and place the results in the odd-numbered 16-bit elements of the destination vector, leaving the even-numbered elements unchanged. Inactive elements in the destination vector register remain unmodified.  
ID\_AA64ZFR0\_EL1.BF16 indicates whether this instruction is implemented.

SVE  
(FEAT\_BF16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	0	0	0	1	0	1	0	1	0	1	Pg			Zn			Zd						

BFCVTNT <Zd>.H, <Pg>/M, <Zn>.S

```
if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, 32) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, 32) then
        bits(32) element = Elem[operand, e, 32];
        Elem[result, 2*e+1, 16] = FPConvertBF(element, FPCR[]);

Z[d, VL] = result;
```

BFDOT (indexed)

BFloat16 floating-point indexed dot product

This instruction delimits the source vectors into pairs of BFloat16 elements. The BFloat16 pairs within the second source vector are specified using an immediate index which selects the same BFloat16 pair position within each 128-bit vector segment. The index range is from 0 to 3.

If FEAT\_EBF16 is not implemented or FPCR.EBF is 0, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second vector. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Disables alternative floating point behaviors, as if FPCR.AH is 0.

If FEAT\_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

- Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as governed by FPCR.RMode, FPCR.FZ, FPCR.AH, and FPCR.FIZ.

Irrespective of FEAT\_EBF16 and FPCR.EBF, this instruction:

- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Generates only the default NaN, as if FPCR.DN is 1.

This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.BF16 indicates whether this instruction is implemented.

SVE
(FEAT\_BF16)

Table with 32 columns (31 to 0) and 2 rows of bit fields: 0 1 1 0 0 1 0 0 0 1 1 i2 Zm 0 1 0 0 0 0 Zn Zda

BFDOT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i2);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index of a group of two 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
constant integer eltspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2 * s + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * s + 1, 16];
    bits(32) sum = Elem[operand3, e, 32];

    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFDOT (vectors)

BFloat16 floating-point dot product

This instruction delimits the source vectors into pairs of BFloat16 elements.

If FEAT\_EBF16 is not implemented or FPCR.EBF is 0, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Disables alternative floating point behaviors, as if FPCR.AH is 0.

If FEAT\_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

- Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as governed by FPCR.RMode, FPCR.FZ, FPCR.AH, and FPCR.FIZ.

Irrespective of FEAT\_EBF16 and FPCR.EBF, this instruction:

- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Generates only the default NaN, as if FPCR.DN is 1.

This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.BF16 indicates whether this instruction is implemented.

SVE
(FEAT\_BF16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	1	0	0	1	0	0	0	1	1	Zm					1	0	0	0	0	0	0	0	Zn					Zda				

BFDOT <Zda>.S, <Zn>.H, <Zm>.H

```
if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
    bits(32) sum = Elem[operand3, e, 32];

    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# BFMAX

BFloat16 floating-point maximum (predicated)

Determine the maximum of active BFloat16 elements of the second source vector and corresponding BFloat16 elements of the first source vector and destructively place the results in the corresponding elements of the first source vector.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

Inactive elements in the destination vector register remain unmodified.

This instruction follows SVE2.1 non-widening BFloat16 numerical behaviors.

ID\_AA64ZFR0\_EL1.B16B16 indicates whether this instruction is implemented.

## SVE2 (FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	0	0	0	0	1	1	0	1	0	0	Pg			Zm			Zdn							
size<1>size<0>																															

**BFMAX** <Zdn>.H, <Pg>/M, <Zdn>.H, <Zm>.H

```
if (!HaveSVE2() && !HaveSME2()) || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

### Assembler Symbols

- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 16;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, 16) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, e, 16];
    if ActivePredicateElement(mask, e, 16) then
        bits(16) element2 = Elem[operand2, e, 16];
        Elem[result, e, 16] = BFMax(element1, element2, FPCR[]);
    else
        Elem[result, e, 16] = element1;

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BFMAXNM

BFloat16 floating-point maximum number (predicated)

Determine the maximum number value of active BFloat16 elements of the second source vector and corresponding BFloat16 elements of the first source vector and destructively place the results in the corresponding elements of the first source vector.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

Inactive elements in the destination vector register remain unmodified.

This instruction follows SVE2.1 non-widening BFloat16 numerical behaviors.

ID\_AA64ZFR0\_EL1.B16B16 indicates whether this instruction is implemented.

## SVE2 (FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	0	0	0	0	0	1	0	0	1	0	0	Pg					Zm					Zdn		
size<1>size<0>																															

**BFMAXNM** <Zdn>.H, <Pg>/M, <Zdn>.H, <Zm>.H

```
if (!HaveSVE2() && !HaveSME2()) || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

## Assembler Symbols

- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 16;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, 16) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, e, 16];
    if ActivePredicateElement(mask, e, 16) then
        bits(16) element2 = Elem[operand2, e, 16];
        Elem[result, e, 16] = BFMaxNum(element1, element2, FPCR[]);
    else
        Elem[result, e, 16] = element1;

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

**BFMIN**

BFloat16 floating-point minimum (predicated)

Determine the minimum of active BFloat16 elements of the second source vector and corresponding BFloat16 elements of the first source vector and destructively place the results in the corresponding elements of the first source vector.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

Inactive elements in the destination vector register remain unmodified.

This instruction follows SVE2.1 non-widening BFloat16 numerical behaviors.

ID\_AA64ZFR0\_EL1.B16B16 indicates whether this instruction is implemented.

**SVE2**  
**(FEAT\_SVE\_B16B16)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	0	0	0	0	0	1	1	1	1	0	0	Pg			Zm				Zdn					
size<1>size<0>																															

**BFMIN** <Zdn>.H, <Pg>/M, <Zdn>.H, <Zm>.H

```
if (!HaveSVE2() && !HaveSME2()) || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

**Assembler Symbols**

- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 16;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, 16) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  bits(16) element1 = Elem[operand1, e, 16];
  if ActivePredicateElement(mask, e, 16) then
    bits(16) element2 = Elem[operand2, e, 16];
    Elem[result, e, 16] = BFMin(element1, element2, FPCR[]);
  else
    Elem[result, e, 16] = element1;

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BFMINNM

BFloat16 floating-point minimum number (predicated)

Determine the minimum number value of active BFloat16 elements of the second source vector and corresponding BFloat16 elements of the first source vector and destructively place the results in the corresponding elements of the first source vector.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

Inactive elements in the destination vector register remain unmodified.

This instruction follows SVE2.1 non-widening BFloat16 numerical behaviors.

ID\_AA64ZFR0\_EL1.B16B16 indicates whether this instruction is implemented.

## SVE2 (FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	0	0	0	0	0	1	0	1	1	0	0	Pg					Zm					Zdn		
size<1>size<0>																															

BFMINNM <Zdn>.H, <Pg>/M, <Zdn>.H, <Zm>.H

```
if (!HaveSVE2() && !HaveSME2()) || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

## Assembler Symbols

- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 16;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, 16) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, e, 16];
    if ActivePredicateElement(mask, e, 16) then
        bits(16) element2 = Elem[operand2, e, 16];
        Elem[result, e, 16] = BFMinNum(element1, element2, FPCR[]);
    else
        Elem[result, e, 16] = element1;

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## BFMLA (indexed)

BFloat16 floating-point fused multiply-add vectors by indexed elements

Multiply all BFloat16 elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively added without intermediate rounding to the corresponding elements of the addend and destination vector.

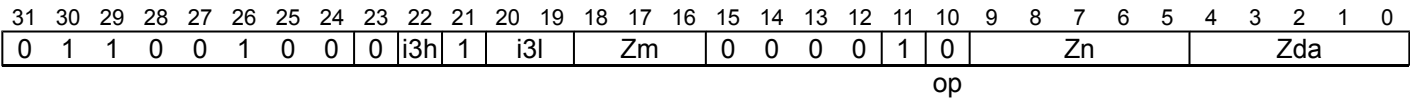
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to 7.

This instruction follows SVE2.1 non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.B16B16 indicates whether this instruction is implemented.

### SVE2 (FEAT\_SVE\_B16B16)



**BFMLA** <Zda>.H, <Zn>.H, <Zm>.H[<imm>]

```
if (!HaveSVE2() && !HaveSME2()) || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 16;
constant integer eltspersegment = 128 DIV 16;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(16) element1 = Elem[operand1, e, 16];
    bits(16) element2 = Elem[operand2, s, 16];
    bits(16) element3 = Elem[result, e, 16];
    if op1_neg then element1 = BFNeg(element1);
    if op3_neg then element3 = BFNeg(element3);
    Elem[result, e, 16] = BFMulAdd(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## BFMLA (vectors)

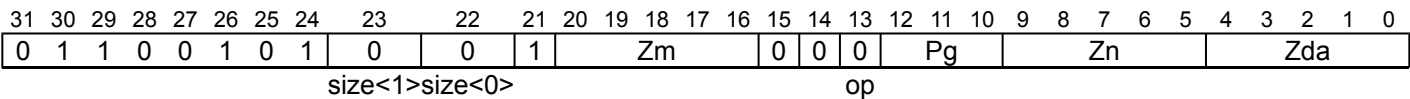
BFloat16 floating-point fused multiply-add vectors

Multiply the corresponding active BFloat16 elements of the first and second source vectors and add to elements of the third source (addend) vector without intermediate rounding. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

This instruction follows SVE2.1 non-widening BFloat16 numerical behaviors.

ID\_AA64ZFR0\_EL1.B16B16 indicates whether this instruction is implemented.

### SVE2 (FEAT\_SVE\_B16B16)



**BFMLA** <Zda>.H, <Pg>/M, <Zn>.H, <Zm>.H

```
if (!HaveSVE2() && !HaveSME2()) || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 16;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, 16) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, 16) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, 16) then
    bits(16) element1 = Elem[operand1, e, 16];
    bits(16) element2 = Elem[operand2, e, 16];
    bits(16) element3 = Elem[operand3, e, 16];

    if op1_neg then element1 = BFNeg(element1);
    if op3_neg then element3 = BFNeg(element3);
    Elem[result, e, 16] = BFMulAdd(element3, element1, element2, FPCR[]);
  else
    Elem[result, e, 16] = Elem[operand3, e, 16];

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

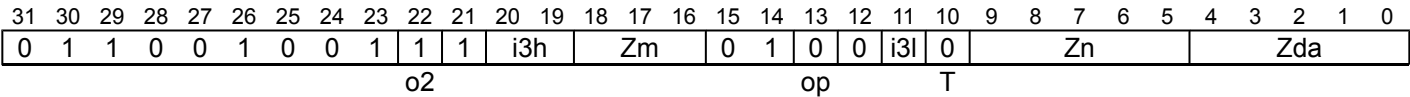
BFMLALB (indexed)

BFloat16 floating-point multiply-add long to single-precision (bottom, indexed)

This BFloat16 floating-point multiply-add long instruction widens the even-numbered BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector. This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.BF16 indicates whether this instruction is implemented.

SVE
(FEAT\_BF16)



BFMLALB <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
constant integer eltspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(16) element1 = Elem[operand1, 2 * e + 0, 16];
    bits(16) element2 = Elem[operand2, s, 16];
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

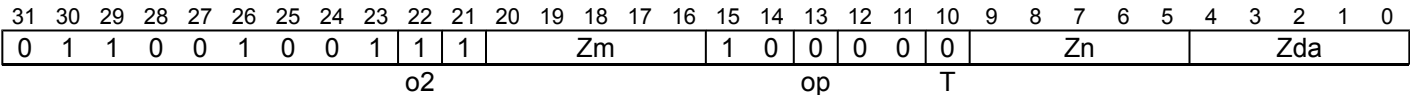
BFMLALB (vectors)

BFloat16 floating-point multiply-add long to single-precision (bottom)

This BFloat16 floating-point multiply-add long instruction widens the even-numbered BFloat16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors. This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.BF16 indicates whether this instruction is implemented.

SVE
(FEAT\_BF16)



BFMLALB <Zda>.S, <Zn>.H, <Zm>.H

```
if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, 2 * e + 0, 16];
    bits(16) element2 = Elem[operand2, 2 * e + 0, 16];
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.





## BFMLALT (indexed)

BFloat16 floating-point multiply-add long to single-precision (top, indexed)

This BFloat16 floating-point multiply-add long instruction widens the odd-numbered BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector. This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.BF16 indicates whether this instruction is implemented.

**SVE**

(FEAT\_BF16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	1	1	i3h		Zm			0	1	0	0	i3l	1				Zn					Zda	
o2										op										T											

**BFMLALT** **<Zda>.S, <Zn>.H, <Zm>.H[<imm>]**

```

if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);

```

## Assembler Symbols

**<Zda>** Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

**<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.

**<Zm>** Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

**<imm>** Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

## Operation

```

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
constant integer eltspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(16) element1 = Elem[operand1, 2 * e + 1, 16];
    bits(16) element2 = Elem[operand2, s, 16];
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;

```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

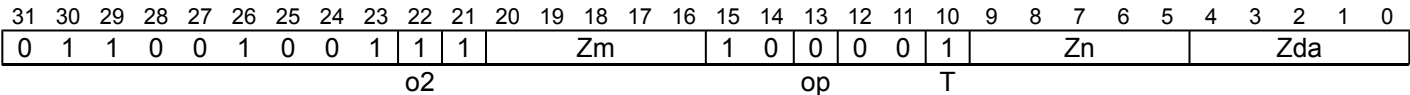
BFMLALT (vectors)

BFloat16 floating-point multiply-add long to single-precision (top)

This BFloat16 floating-point multiply-add long instruction widens the odd-numbered BFloat16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors. This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.BF16 indicates whether this instruction is implemented.

SVE
(FEAT\_BF16)



BFMLALT <Zda>.S, <Zn>.H, <Zm>.H

```
if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, 2 * e + 1, 16];
    bits(16) element2 = Elem[operand2, 2 * e + 1, 16];
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



## BFMLS (indexed)

BFloat16 floating-point fused multiply-subtract vectors by indexed elements

Multiply all BFloat16 elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively subtracted without intermediate rounding from the corresponding elements of the addend and destination vector.

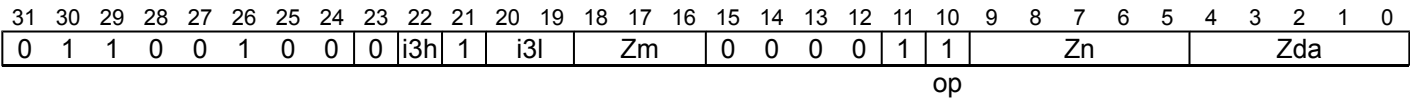
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to 7.

This instruction follows SVE2.1 non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.B16B16 indicates whether this instruction is implemented.

### SVE2 (FEAT\_SVE\_B16B16)



**BFMLS** <Zda>.H, <Zn>.H, <Zm>.H[<imm>]

```
if (!HaveSVE2() && !HaveSME2()) || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 16;
constant integer eltspersegment = 128 DIV 16;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(16) element1 = Elem[operand1, e, 16];
    bits(16) element2 = Elem[operand2, s, 16];
    bits(16) element3 = Elem[result, e, 16];
    if op1_neg then element1 = BFNeg(element1);
    if op3_neg then element3 = BFNeg(element3);
    Elem[result, e, 16] = BFMulAdd(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

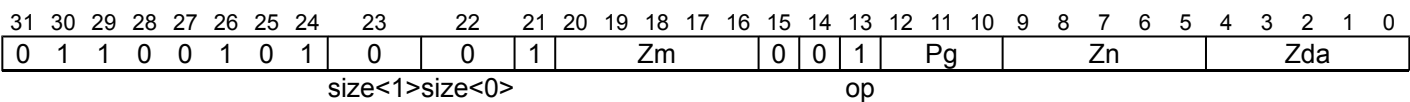
## BFMLS (vectors)

BFloat16 floating-point fused multiply-subtract vectors

Multiply the corresponding active BFloat16 elements of the first and second source vectors and subtract from elements of the third source (addend) vector without intermediate rounding. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified. This instruction follows SVE2.1 non-widening BFloat16 numerical behaviors.

ID\_AA64ZFR0\_EL1.B16B16 indicates whether this instruction is implemented.

### SVE2 (FEAT\_SVE\_B16B16)



**BFMLS** <Zda>.H, <Pg>/M, <Zn>.H, <Zm>.H

```
if (!HaveSVE2() && !HaveSME2()) || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 16;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, 16) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, 16) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, 16) then
    bits(16) element1 = Elem[operand1, e, 16];
    bits(16) element2 = Elem[operand2, e, 16];
    bits(16) element3 = Elem[operand3, e, 16];

    if op1_neg then element1 = BFNeg(element1);
    if op3_neg then element3 = BFNeg(element3);
    Elem[result, e, 16] = BFMulAdd(element3, element1, element2, FPCR[]);
  else
    Elem[result, e, 16] = Elem[operand3, e, 16];

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

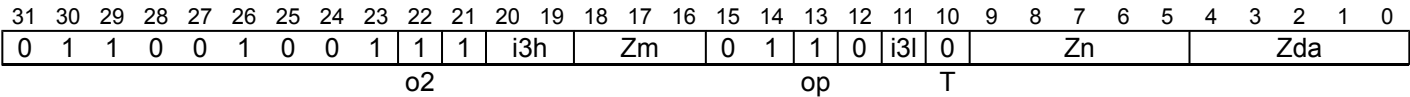


## BFMLSLB (indexed)

BFloat16 floating-point multiply-subtract long from single-precision (bottom, indexed)

This BFloat16 floating-point multiply-subtract long instruction widens the even-numbered BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector. This instruction is unpredicated.

### SVE2 (FEAT\_SVE2p1)



```
BFMLSLB <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
boolean op1_neg = TRUE;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
constant integer eltspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(16) element1 = Elem[operand1, 2 * e + 0, 16];
    bits(16) element2 = Elem[operand2, s, 16];
    bits(32) element3 = Elem[operand3, e, 32];
    if op1_neg then element1 = BFNeg(element1);
    Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

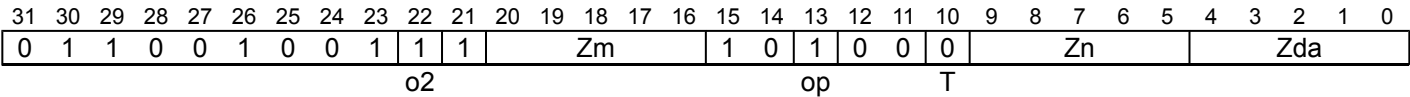
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## BFMLSLB (vectors)

BFloat16 floating-point multiply-subtract long from single-precision (bottom)

This BFloat16 floating-point multiply-subtract long instruction widens the even-numbered BFloat16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors. This instruction is unpredicated.

### SVE2 (FEAT\_SVE2p1)



```
BFMLSLB <Zda>.S, <Zn>.H, <Zm>.H
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, 2 * e + 0, 16];
    bits(16) element2 = Elem[operand2, 2 * e + 0, 16];
    bits(32) element3 = Elem[operand3, e, 32];
    if op1_neg then element1 = BFNeg(element1);
    Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

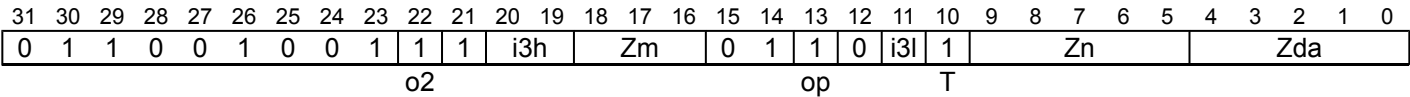


## BFMLSLT (indexed)

BFloat16 floating-point multiply-subtract long from single-precision (top, indexed)

This BFloat16 floating-point multiply-subtract long instruction widens the odd-numbered BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector. This instruction is unpredicated.

### SVE2 (FEAT\_SVE2p1)



**BFMLSLT** <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
boolean opl_neg = TRUE;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
constant integer eltspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(16) element1 = Elem[operand1, 2 * e + 1, 16];
    bits(16) element2 = Elem[operand2, s, 16];
    bits(32) element3 = Elem[operand3, e, 32];
    if opl_neg then element1 = BFNeg(element1);
    Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

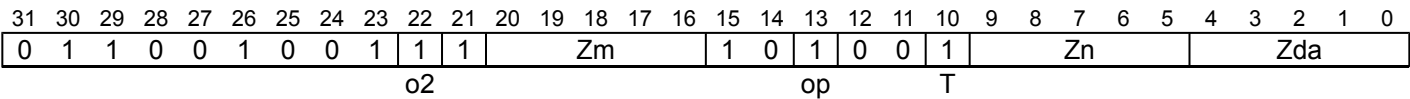
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## BFMLSLT (vectors)

BFloat16 floating-point multiply-subtract long from single-precision (top)

This BFloat16 floating-point multiply-subtract long instruction widens the odd-numbered BFloat16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors. This instruction is unpredicated.

### SVE2 (FEAT\_SVE2p1)



```
BFMLSLT <Zda>.S, <Zn>.H, <Zm>.H
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, 2 * e + 1, 16];
    bits(16) element2 = Elem[operand2, 2 * e + 1, 16];
    bits(32) element3 = Elem[operand3, e, 32];
    if op1_neg then element1 = BFNeg(element1);
    Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.





BFMMLA

BFloat16 floating-point matrix multiply-accumulate into 2x2 matrices

If FEAT\_EBF16 is not implemented or FPCR.EBF is 0, this instruction:

- Performs two unfused sums-of-products within each two pairs of adjacent BFloat16 elements while multiplying the 2x4 matrix of BFloat16 values held in each 128-bit segment of the first source vector by the 4x2 matrix of BFloat16 values in the corresponding segment of the second source vector. The intermediate single-precision products are rounded before they are summed and the intermediate sum is rounded before accumulation into the 2x2 single-precision matrix in the corresponding segment of the destination vector. This is equivalent to accumulating two 2-way unfused dot products per destination element.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Disables alternative floating point behaviors, as if FPCR.AH is 0.

If FEAT\_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

- Performs two fused sums-of-products within each two pairs of adjacent BFloat16 elements while multiplying the 2x4 matrix of BFloat16 values held in each 128-bit segment of the first source vector by the 4x2 matrix of BFloat16 values in the corresponding segment of the second source vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the 2x2 single-precision matrix in the corresponding segment of the destination vector. This is equivalent to accumulating two 2-way fused dot products per destination element.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as governed by FPCR.RMode, FPCR.FZ, FPCR.AH, and FPCR.FIZ.

Irrespective of FEAT\_EBF16 and FPCR.EBF, this instruction:

- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Generates only the default NaN, as if FPCR.DN is 1.

This instruction is unpredicated and vector length agnostic.

ID\_AA64ZFR0\_EL1.BF16 indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE
(FEAT\_BF16)

Table with 32 columns (31 to 0) and 1 row. Columns 31-24 contain 0s. Columns 23-21 contain 0, 1, 1. Columns 20-16 contain Zm. Columns 15-10 contain 1, 1, 1, 0, 0, 1. Columns 9-6 contain Zn. Columns 5-0 contain Zda.

BFMMLA <Zda>.S, <Zn>.H, <Zm>.H

```
if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
bits(128) op1, op2;
bits(128) res, addend;

for s = 0 to segments-1
    op1    = Elem[operand1, s, 128];
    op2    = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];
    res    = BFMatMulAdd(addend, op1, op2);
    Elem[result, s, 128] = res;

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFMUL (indexed)

BFloat16 floating-point multiply vectors by indexed elements

Multiply all BFloat16 elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment and place the results in the corresponding elements of the destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to 7.

This instruction follows SVE2.1 non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.B16B16 indicates whether this instruction is implemented.

SVE2
(FEAT\_SVE\_B16B16)

Table with 32 columns (31 to 0) and 2 rows. Row 1 contains bit positions. Row 2 contains fields: 0, 1, 1, 0, 0, 1, 0, 0, 0, i3h, 1, i3l, Zm, 0, 0, 1, 0, 1, 0, Zn, Zd.

BFMUL <Zd>.H, <Zn>.H, <Zm>.H[<imm>]

```
if (!HaveSVE2() && !HaveSME2()) || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 16;
constant integer eltspersegment = 128 DIV 16;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(16) element1 = Elem[operand1, e, 16];
    bits(16) element2 = Elem[operand2, s, 16];
    Elem[result, e, 16] = BFMul(element1, element2, FPCR[]);

Z[d, VL] = result;
```

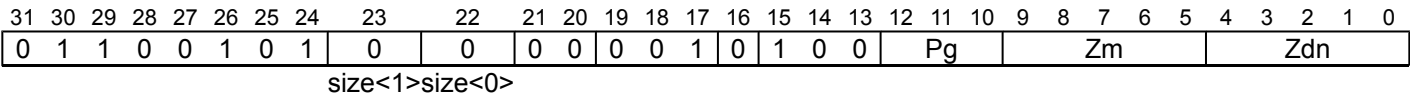
BFMUL (vectors, predicated)

BFloat16 floating-point multiply vectors (predicated)

Multiply active BFloat16 elements of the second source vector to corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

This instruction follows SVE2.1 non-widening BFloat16 numerical behaviors.  
ID\_AA64ZFR0\_EL1.B16B16 indicates whether this instruction is implemented.

SVE2  
(FEAT\_SVE\_B16B16)



BFMUL <Zdn>.H, <Pg>/M, <Zdn>.H, <Zm>.H

```
if (!HaveSVE2() && !HaveSME2()) || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 16;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, 16) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  bits(16) element1 = Elem[operand1, e, 16];
  if ActivePredicateElement(mask, e, 16) then
    bits(16) element2 = Elem[operand2, e, 16];
    Elem[result, e, 16] = BFMul(element1, element2, FPCR[]);
  else
    Elem[result, e, 16] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

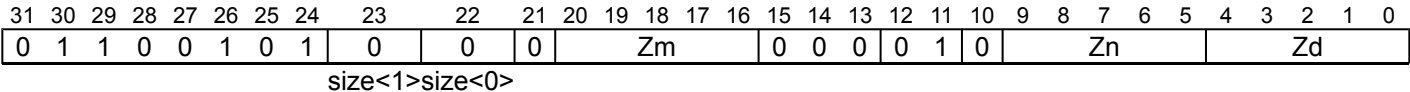


BFMUL (vectors, unpredicated)

BFloat16 floating-point multiply vectors (unpredicated)

Multiply all BFloat16 elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector.  
This instruction follows SVE2.1 non-widening BFloat16 numerical behaviors.  
This instruction is unpredicated.  
ID\_AA64ZFR0\_EL1.B16B16 indicates whether this instruction is implemented.

SVE2  
(FEAT\_SVE\_B16B16)



BFMUL <Zd>.H, <Zn>.H, <Zm>.H

```
if (!HaveSVE2() && !HaveSME2()) || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, e, 16];
    bits(16) element2 = Elem[operand2, e, 16];
    Elem[result, e, 16] = BFMul(element1, element2, FPCR[]);

Z[d, VL] = result;
```

## BFSUB (predicated)

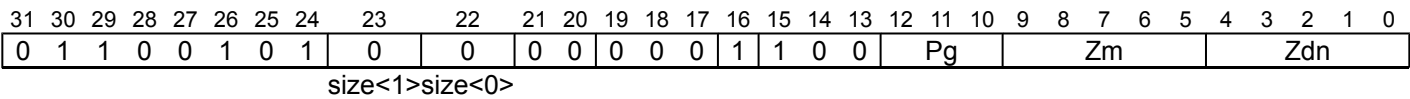
BFloat16 floating-point subtract vectors (predicated)

Subtract active BFloat16 elements of the second source vector from corresponding BFloat16 elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

This instruction follows SVE2.1 non-widening BFloat16 numerical behaviors.

ID\_AA64ZFR0\_EL1.B16B16 indicates whether this instruction is implemented.

### SVE2 (FEAT\_SVE\_B16B16)



**BFSUB** <Zdn>.H, <Pg>/M, <Zdn>.H, <Zm>.H

```
if (!HaveSVE2() && !HaveSME2()) || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

### Assembler Symbols

- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 16;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, 16) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, e, 16];
    if ActivePredicateElement(mask, e, 16) then
        bits(16) element2 = Elem[operand2, e, 16];
        Elem[result, e, 16] = BFSUB(element1, element2, FPCR[]);
    else
        Elem[result, e, 16] = element1;

Z[dn, VL] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.





BFSUB (unpredicated)

BFloat16 floating-point subtract vectors (unpredicated)

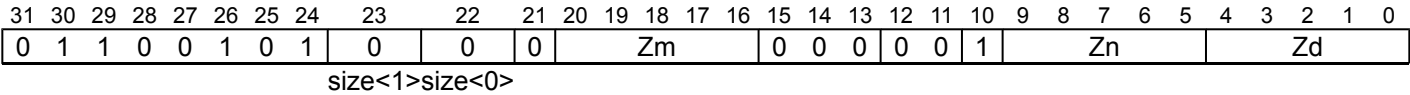
Subtract all BFloat16 elements of the second source vector from corresponding BFloat16 elements of the first source vector and place the results in the corresponding elements of the destination vector.

This instruction follows SVE2.1 non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.B16B16 indicates whether this instruction is implemented.

SVE2
(FEAT\_SVE\_B16B16)



BFSUB <Zd>.H, <Zn>.H, <Zm>.H

```
if (!HaveSVE2() && !HaveSME2()) || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, e, 16];
    bits(16) element2 = Elem[operand2, e, 16];
    Elem[result, e, 16] = BFSUB(element1, element2, FPCR[]);

Z[d, VL] = result;
```

BGRP

Group bits to right or left as selected by bitmask

This instruction separates bits in each element of the first source vector by gathering from the bit positions indicated by non-zero bits in the corresponding mask element of the second source vector to the lowest-numbered contiguous bits of the corresponding destination element, and from positions indicated by zero bits to the highest-numbered bits of the destination element, preserving the bit order within each group. This instruction is unpredicated.  
ID\_AA64ZFR0\_EL1.BitPerm indicates whether this instruction is implemented.  
This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE2  
(FEAT\_SVE\_BitPerm)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size		0	Zm				1	0	1	1	1	0	Zn				Zd						

BGRP <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() || !HaveSVE2BitPerm() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) data = Z[n, VL];
bits(VL) mask = Z[m, VL];
bits(VL) result;

for e = 0 to elements - 1
    Elem[result, e, esize] = BitGroup(Elem[data, e, esize], Elem[mask, e, esize]);

Z[d, VL] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BIC (immediate)

Bitwise clear bits using immediate (unpredicated)

Bitwise clear bits using immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This is a pseudo-instruction of [AND \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [AND \(immediate\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [AND \(immediate\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	1	0	0	0	0	0	imm13														Zdn			

BIC <Zdn>.<T>, <Zdn>.<T>, #<const>

is equivalent to

AND <Zdn>.<T>, <Zdn>.<T>, #(-<const> - 1)

## Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "imm13<12>:imm13<5:0>":

imm13<12>	imm13<5:0>	<T>
0	0xxxxx	S
0	10xxxx	H
0	110xxx	B
0	1110xx	B
0	11110x	B
0	111110	RESERVED
0	111111	RESERVED
1	xxxxxx	D

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

## Operation

The description of [AND \(immediate\)](#) gives the operational pseudocode for this instruction.

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



## BIC (predicates)

Bitwise clear predicates

Bitwise AND inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	0	0	0	Pm				0	1	Pg				0	Pn				1	Pd			
S																															

**BIC** <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

### Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit element1 = PredicateElement(operand1, e, esize);
    bit element2 = PredicateElement(operand2, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, psize] = ZeroExtend(element1 AND (NOT element2), psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

### Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BIC (vectors, predicated)

Bitwise clear vectors (predicated)

Bitwise AND inverted active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	0	1	1	0	0	0	Pg													

BIC <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

## Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = element1 AND (NOT element2);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:



- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BIC (vectors, unpredicated)

Bitwise clear vectors (unpredicated)

Bitwise AND inverted all elements of the second source vector with corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	1	1	Zm				0	0	1	1	0	0	Zn				Zd						

BIC <Zd>.D, <Zn>.D, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

## Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];

Z[d, VL] = operand1 AND (NOT operand2);
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## BICS

Bitwise clear predicates, setting the condition flags

Bitwise AND inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	1	0	0	Pm			0	1	Pg			0	Pn			1	Pd						
S																															

**BICS** <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

### Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit element1 = PredicateElement(operand1, e, esize);
    bit element2 = PredicateElement(operand2, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, psize] = ZeroExtend(element1 AND (NOT element2), psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

### Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BRKA

Break after first true condition

Sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected. Does not set the condition flags.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	0	0	1	0	0	0	0	0	1	Pg			0	Pn			M	Pd					
								B		S																					

BRKA <Pd>.B, <Pg>/<ZM>, <Pn>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = (M == '1');
boolean setflags = FALSE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <ZM> Is the predication qualifier, encoded in "M":

M	<ZM>
0	Z
1	M

- <Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand = P[n, PL];
bits(PL) operand2 = P[d, PL];
boolean break = FALSE;
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    boolean element = ActivePredicateElement(operand, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        bit pbit = if !break then '1' else '0';
        Elem[result, e, psize] = ZeroExtend(pbit, psize);
        break = break || element;
    elsif merging then
        bit pbit = PredicateElement(operand2, e, esize);
        Elem[result, e, psize] = ZeroExtend(pbit, psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```



BRKAS

Break after first true condition, setting the condition flags

Sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	1	0	1	0	0	0	0	0	1	Pg			0	Pn			0	Pd					
B										S																	M				

BRKAS <Pd>.B, <Pg>/Z, <Pn>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = FALSE;
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand = P[n, PL];
bits(PL) operand2 = P[d, PL];
boolean break = FALSE;
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    boolean element = ActivePredicateElement(operand, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        bit pbit = if !break then '1' else '0';
        Elem[result, e, psize] = ZeroExtend(pbit, psize);
        break = break || element;
    elsif merging then
        bit pbit = PredicateElement(operand2, e, esize);
        Elem[result, e, psize] = ZeroExtend(pbit, psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.





BRKB

Break before first true condition

Sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected. Does not set the condition flags.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	1	0	0	1	0	0	0	0	0	1	Pg			0	Pn			M	Pd					
										B		S																			

BRKB <Pd>.B, <Pg>/<ZM>, <Pn>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = (M == '1');
boolean setflags = FALSE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <ZM> Is the predication qualifier, encoded in "M":

M	<ZM>
0	Z
1	M

- <Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand = P[n, PL];
bits(PL) operand2 = P[d, PL];
boolean break = FALSE;
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    boolean element = ActivePredicateElement(operand, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        break = break || element;
        bit pbit = if !break then '1' else '0';
        Elem[result, e, psize] = ZeroExtend(pbit, psize);
    elsif merging then
        bit pbit = PredicateElement(operand2, e, esize);
        Elem[result, e, psize] = ZeroExtend(pbit, psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```



# BRKBS

Break before first true condition, setting the condition flags

Sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	1	1	0	1	0	0	0	0	0	1	Pg			0	Pn			0	Pd					
B										S																	M				

BRKBS <Pd>.B, <Pg>/Z, <Pn>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = FALSE;
boolean setflags = TRUE;
```

## Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand = P[n, PL];
bits(PL) operand2 = P[d, PL];
boolean break = FALSE;
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    boolean element = ActivePredicateElement(operand, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        break = break || element;
        bit pbit = if !break then '1' else '0';
        Elem[result, e, psize] = ZeroExtend(pbit, psize);
    elsif merging then
        bit pbit = PredicateElement(operand2, e, esize);
        Elem[result, e, psize] = ZeroExtend(pbit, psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.



BRKN

Propagate break to next partition

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise leaves the destination and second source predicate unchanged. Does not set the condition flags.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	0	1	0	1	0	0	0	1	1	0	0	0	0	1	Pg			0	Pn			0	Pdm						
S																																

BRKN <Pdm>.B, <Pg>/Z, <Pn>.B, <Pdm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer dm = UInt(Pdm);
boolean setflags = FALSE;
```

Assembler Symbols

- <Pdm> Is the name of the second source and destination scalable predicate register, encoded in the "Pdm" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[dm, PL];
bits(PL) result;

if LastActive(mask, operand1, 8) == '1' then
    result = operand2;
else
    result = Zeros(PL);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(Ones(PL), result, 8);
P[dm, PL] = result;
```

BRKNS

Propagate break to next partition, setting the condition flags

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise leaves the destination and second source predicate unchanged. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	0	1	0	1	0	1	0	1	1	0	0	0	0	1	Pg			0	Pn			0	Pdm						
S																																

BRKNS <Pdm>.B, <Pg>/Z, <Pn>.B, <Pdm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer dm = UInt(Pdm);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pdm> Is the name of the second source and destination scalable predicate register, encoded in the "Pdm" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[dm, PL];
bits(PL) result;

if LastActive(mask, operand1, 8) == '1' then
    result = operand2;
else
    result = Zeros(PL);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(Ones(PL), result, 8);
P[dm, PL] = result;
```

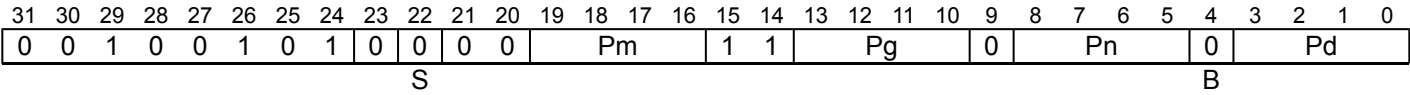
Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

# BRKPA

Break after first true condition, propagating from previous partition

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.



BRKPA <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

## Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, 8) then
        bit pbit = if last then '1' else '0';
        Elem[result, e, 1] = ZeroExtend(pbit, 1);
        last = last && (!ActivePredicateElement(operand2, e, 8));
    else
        Elem[result, e, 1] = ZeroExtend('0', 1);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

# BRKPAS

Break after first true condition, propagating from previous partition and setting the condition flags

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	1	0	0	Pm			1	1	Pg			0	Pn			0	Pd						
S												B																			

BRKPAS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

## Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, 8) then
    bit pbit = if last then '1' else '0';
    Elem[result, e, 1] = ZeroExtend(pbit, 1);
    last = last && (!ActivePredicateElement(operand2, e, 8));
  else
    Elem[result, e, 1] = ZeroExtend('0', 1);

if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

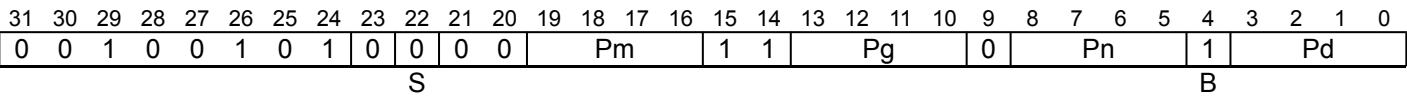
If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.



BRKPB

Break before first true condition, propagating from previous partition

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.



BRKPB <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');

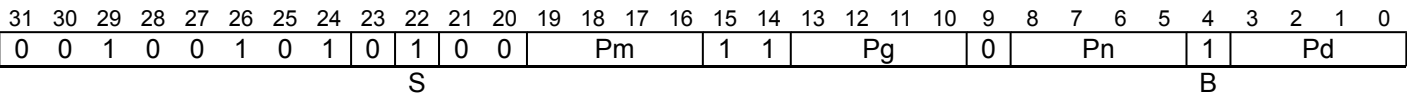
for e = 0 to elements-1
  if ActivePredicateElement(mask, e, 8) then
    last = last && (!ActivePredicateElement(operand2, e, 8));
    bit pbit = if last then '1' else '0';
    Elem[result, e, 1] = ZeroExtend(pbit, 1);
  else
    Elem[result, e, 1] = ZeroExtend('0', 1);

if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

BRKPBS

Break before first true condition, propagating from previous partition and setting the condition flags

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



BRKPBS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, 8) then
    last = last && (!ActivePredicateElement(operand2, e, 8));
    bit pbit = if last then '1' else '0';
    Elem[result, e, 1] = ZeroExtend(pbit, 1);
  else
    Elem[result, e, 1] = ZeroExtend('0', 1);

if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

BSL

Bitwise select

Selects bits from the first source vector where the corresponding bit in the third source vector is '1', and from the second source vector where the corresponding bit in the third source vector is '0'. The result is placed destructively in the destination and first source vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	1					Zm	0	0	1	1	1	1				Zk				Zdn		

```
BSL <Zdn>.D, <Zdn>.D, <Zm>.D, <Zk>.D
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);
```

Assembler Symbols

- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <Zk> Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[k, VL];

Z[dn, VL] = (operand1 AND operand3) OR (operand2 AND NOT(operand3));
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

BSL1N

Bitwise select with first input inverted

Selects bits from the inverted first source vector where the corresponding bit in the third source vector is '1', and from the second source vector where the corresponding bit in the third source vector is '0'. The result is placed destructively in the destination and first source vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	1	1	Zm					0	0	1	1	1	1	Zk					Zdn				

```
BSL1N <Zdn>.D, <Zdn>.D, <Zm>.D, <Zk>.D
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);
```

Assembler Symbols

- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <Zk> Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[k, VL];

Z[dn, VL] = (NOT(operand1) AND operand3) OR (operand2 AND NOT(operand3));
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

BSL2N

Bitwise select with second input inverted

Selects bits from the first source vector where the corresponding bit in the third source vector is '1', and from the inverted second source vector where the corresponding bit in the third source vector is '0'. The result is placed destructively in the destination and first source vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	0	1	Zm					0	0	1	1	1	1	Zk					Zdn				

```
BSL2N <Zdn>.D, <Zdn>.D, <Zm>.D, <Zk>.D
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);
```

Assembler Symbols

- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <Zk> Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[k, VL];

Z[dn, VL] = (operand1 AND operand3) OR (NOT(operand2) AND NOT(operand3));
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

CADD

Complex integer add with rotate

Add the real and imaginary components of the integral complex numbers from the first source vector to the complex numbers from the second source vector which have first been rotated by 90 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, equivalent to multiplying the complex numbers in the second source vector by  $\pm j$  beforehand. Destructively place the results in the corresponding elements of the first source vector. This instruction is unpredicated.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size	0	0	0	0	0	0	0	1	1	0	1	1	rot	Zm				Zdn					

CADD <Zdn>.<T>, <Zdn>.<T>, <Zm>.<T>, <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
boolean sub_i = (rot == '0');
boolean sub_r = (rot == '1');
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in "rot":

rot	<const>
0	#90
1	#270

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for p = 0 to pairs-1
    integer acc_r = SInt(Elem[operand1, 2 * p + 0, esize]);
    integer acc_i = SInt(Elem[operand1, 2 * p + 1, esize]);
    integer elt2_r = SInt(Elem[operand2, 2 * p + 0, esize]);
    integer elt2_i = SInt(Elem[operand2, 2 * p + 1, esize]);
    if sub_i then
        acc_r = acc_r - elt2_i;
        acc_i = acc_i + elt2_r;
    if sub_r then
        acc_r = acc_r + elt2_i;
        acc_i = acc_i - elt2_r;

    Elem[result, 2 * p + 0, esize] = acc_r<esize-1:0>;
    Elem[result, 2 * p + 1, esize] = acc_i<esize-1:0>;

Z[dn, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CDOT (indexed)

### Complex integer dot product (indexed)

The complex integer dot product instructions delimit the source vectors into pairs of 8-bit or 16-bit signed integer complex numbers. Within each pair, the complex numbers in the first source vector are multiplied by the corresponding complex numbers in the second source vector and the resulting wide real or wide imaginary part of the product is accumulated into a 32-bit or 64-bit destination vector element which overlaps all four of the elements that comprise a pair of complex number values in the first source vector.

As a result each instruction implicitly deinterleaves the real and imaginary components of their complex number inputs, so that the destination vector accumulates 4×wide real sums or 4×wide imaginary sums. The complex numbers in the second source vector are rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, by performing the following transformations prior to the dot product operations:

- If the rotation is #0, the imaginary parts of the complex numbers in the second source vector are negated. The destination vector therefore accumulates the real parts of a complex dot product.
- If the rotation is #90, the real and imaginary parts of the complex numbers the second source vector are swapped. The destination vector therefore accumulates the imaginary parts of a complex dot product.
- If the rotation is #180, there is no transformation. The destination vector therefore accumulates the real parts of a complex conjugate dot product.
- If the rotation is #270, the real parts of the complex numbers in the second source vector are negated and then swapped with the imaginary parts. The destination vector therefore accumulates the imaginary parts of a complex conjugate dot product.

The indexed form of these instructions select a single pair of complex numbers within each 128-bit segment of the second source vector as the multiplier for all pairs of complex numbers within the corresponding 128-bit segment of the first source vector. The complex number pairs within the second source vector are specified using an immediate index which selects the same complex number pair position within each 128-bit vector segment. The index range is from 0 to one less than the number of complex number pairs per 128-bit segment, encoded in 1 or 2 bits depending on the size of the complex number pair.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

### 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	0	1	i2		Zm			0	1	0	0	rot				Zn					Zda		
size<1>size<0>																															

CDOT <Zda>.S, <Zn>.B, <Zm>.B[<imm>], <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_i = (rot<0> == rot<1>);
```

### 64-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	1	1	i1		Zm			0	1	0	0	rot				Zn					Zda		
size<1>size<0>																															



CDOT [<Zda>.D](#), [<Zn>.H](#), [<Zm>.H\[<imm>\]](#), [<const>](#)

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_i = (rot<0> == rot<1>);
```

Assembler Symbols

- [<Zda>](#) Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- [<Zn>](#) Is the name of the first source scalable vector register, encoded in the "Zn" field.
- [<Zm>](#) For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- [<imm>](#) For the 32-bit variant: is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.  
For the 64-bit variant: is the immediate index of a 64-bit group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the "i1" field.
- [<const>](#) Is the const specifier, encoded in "rot":

rot	<const>
00	#0
01	#90
10	#180
11	#270

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 1
        integer elt1_r = SInt(Elem[operand1, 4 * e + 2 * i + 0, esize DIV 4]);
        integer elt1_i = SInt(Elem[operand1, 4 * e + 2 * i + 1, esize DIV 4]);
        integer elt2_a = SInt(Elem[operand2, 4 * s + 2 * i + sel_a, esize DIV 4]);
        integer elt2_b = SInt(Elem[operand2, 4 * s + 2 * i + sel_b, esize DIV 4]);
        if sub_i then
            res = (res + (elt1_r * elt2_a)) - (elt1_i * elt2_b);
        else
            res = res + (elt1_r * elt2_a) + (elt1_i * elt2_b);
        Elem[result, e, esize] = res;

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CDOT (vectors)

Complex integer dot product

The complex integer dot product instructions delimit the source vectors into pairs of 8-bit or 16-bit signed integer complex numbers. Within each pair, the complex numbers in the first source vector are multiplied by the corresponding complex numbers in the second source vector and the resulting wide real or wide imaginary part of the product is accumulated into a 32-bit or 64-bit destination vector element which overlaps all four of the elements that comprise a pair of complex number values in the first source vector.

As a result each instruction implicitly deinterleaves the real and imaginary components of their complex number inputs, so that the destination vector accumulates 4×wide real sums or 4×wide imaginary sums.

The complex numbers in the second source vector are rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, by performing the following transformations prior to the dot product operations:

- If the rotation is #0, the imaginary parts of the complex numbers in the second source vector are negated. The destination vector therefore accumulates the real parts of a complex dot product.
- If the rotation is #90, the real and imaginary parts of the complex numbers the second source vector are swapped. The destination vector therefore accumulates the imaginary parts of a complex dot product.
- If the rotation is #180, there is no transformation. The destination vector therefore accumulates the real parts of a complex conjugate dot product.
- If the rotation is #270, the real parts of the complex numbers in the second source vector are negated and then swapped with the imaginary parts. The destination vector therefore accumulates the imaginary parts of a complex conjugate dot product.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	0	Zm				0	0	0	1	rot	Zn				Zda								

CDOT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>, <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_i = (rot<0> == rot<1>);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size<0>":

size<0>	<T>
0	S
1	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size<0>":

size<0>	<Tb>
0	B
1	H

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in "rot":

rot	<const>
00	#0
01	#90
10	#180
11	#270

## Operation

```

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 1
        integer elt1_r = SInt(Elem[operand1, 4 * e + 2 * i + 0, esize DIV 4]);
        integer elt1_i = SInt(Elem[operand1, 4 * e + 2 * i + 1, esize DIV 4]);
        integer elt2_a = SInt(Elem[operand2, 4 * e + 2 * i + sel_a, esize DIV 4]);
        integer elt2_b = SInt(Elem[operand2, 4 * e + 2 * i + sel_b, esize DIV 4]);
        if sub_i then
            res = (res + (elt1_r * elt2_a)) - (elt1_i * elt2_b);
        else
            res = res + (elt1_r * elt2_a) + (elt1_i * elt2_b);
        Elem[result, e, esize] = res;
Z[da, VL] = result;

```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

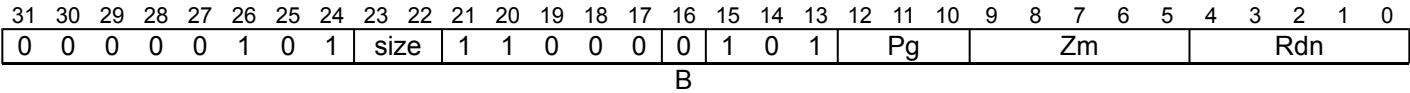
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CLASTA (scalar)

Conditionally extract element after last to general-purpose register

From the source vector register extract the element after the last active element, or if the last active element is the final element extract element zero, and then zero-extend that element to destructively place in the destination and first source general-purpose register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source general-purpose register.



```
CLASTA <R><dn>, <Pg>, <R><dn>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Rdn);
integer m = UInt(Zm);
constant integer csize = if esize < 64 then 32 else 64;
boolean isBefore = FALSE;
```

Assembler Symbols

<R> Is a width specifier, encoded in “size”:

size	<R>
01	W
x0	W
11	X

<dn> Is the number [0-30] of the source and destination general-purpose register or the name ZR (31), encoded in the "Rdn" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the source scalable vector register, encoded in the "Zm" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(esize) operand1 = X[dn, esize];
bits(VL) operand2 = Z[m, VL];
bits(csize) result;
integer last = LastActiveElement(mask, esize);

if last < 0 then
    result = ZeroExtend(operand1, csize);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
    result = ZeroExtend(Elem[operand2, last, esize], csize);

X[dn, csize] = result;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CLASTA (SIMD&FP scalar)

Conditionally extract element after last to SIMD&FP scalar register

From the source vector register extract the element after the last active element, or if the last active element is the final element extract element zero, and then zero-extend that element to destructively place in the destination and first source SIMD & floating-point scalar register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source SIMD & floating-point scalar register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	0	1	0	1	0	1	0	0	0	Pg	Zm					Vdn						

B

CLASTA <V><dn>, <Pg>, <V><dn>, <Zm>.<T>

```

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Vdn);
integer m = UInt(Zm);
boolean isBefore = FALSE;

```

## Assembler Symbols

**<V>** Is a width specifier, encoded in "size":

size	$\langle V \rangle$
00	B
01	H
10	S
11	D

<dn> Is the number [0-31] of the source and destination SIMD&FP register, encoded in the "Vdn" field.

**<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

**<Zm>** Is the name of the source scalable vector register, encoded in the "Zm" field.

**<T>** Is the size specifier, encoded in "size":

size	$\langle T \rangle$
00	B
01	H
10	S
11	D

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(esize) operand1 = V[dn, esize];
bits(VL) operand2 = Z[m, VL];
bits(esize) result;
integer last = LastActiveElement(mask, esize);

if last < 0 then
    result = ZeroExtend(operand1, esize);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
    result = Elem[operand2, last, esize];

V[dn, esize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



CLASTA (vectors)

Conditionally extract element after last to vector register

From the second source vector register extract the element after the last active element, or if the last active element is the final element extract element zero, and then replicate that element to destructively fill the destination and first source vector.

If there are no active elements then leave the destination and source vector unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	0	1	0	0	0	0	1	0	0	Pg	Zm			Zdn								
B																															

CLASTA <Zdn>.<T>, <Pg>, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean isBefore = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
integer last = LastActiveElement(mask, esize);

if last < 0 then
    result = operand1;
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
    for e = 0 to elements-1
        Elem[result, e, esize] = Elem[operand2, last, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

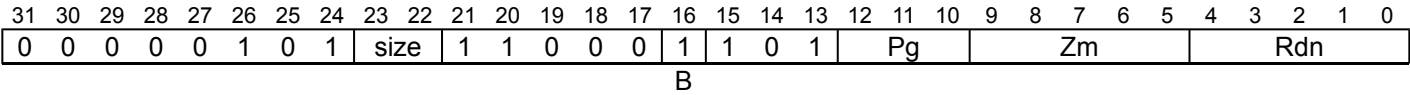
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CLASTB (scalar)

Conditionally extract last element to general-purpose register

From the source vector register extract the last active element, and then zero-extend that element to destructively place in the destination and first source general-purpose register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source general-purpose register.



```
CLASTB <R><dn>, <Pg>, <R><dn>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Rdn);
integer m = UInt(Zm);
constant integer csize = if esize < 64 then 32 else 64;
boolean isBefore = TRUE;
```

Assembler Symbols

<R> Is a width specifier, encoded in “size”:

size	<R>
01	W
x0	W
11	X

<dn> Is the number [0-30] of the source and destination general-purpose register or the name ZR (31), encoded in the "Rdn" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the source scalable vector register, encoded in the "Zm" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(esize) operand1 = X[dn, esize];
bits(VL) operand2 = Z[m, VL];
bits(csize) result;
integer last = LastActiveElement(mask, esize);

if last < 0 then
    result = ZeroExtend(operand1, csizesize);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
    result = ZeroExtend(Elem[operand2, last, esize], csizesize);

X[dn, csizesize] = result;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

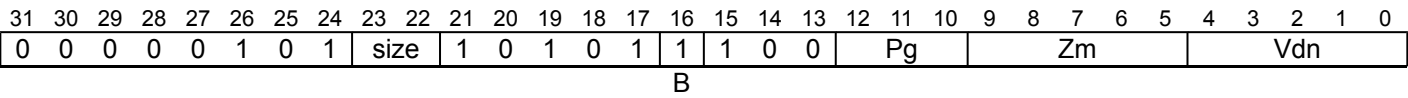
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CLASTB (SIMD&FP scalar)

Conditionally extract last element to SIMD&FP scalar register

From the source vector register extract the last active element, and then zero-extend that element to destructively place in the destination and first source SIMD & floating-point scalar register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source SIMD & floating-point scalar register.



```
CLASTB <V><dn>, <Pg>, <V><dn>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Vdn);
integer m = UInt(Zm);
boolean isBefore = TRUE;
```

Assembler Symbols

<V> Is a width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	D

<dn> Is the number [0-31] of the source and destination SIMD&FP register, encoded in the "Vdn" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the source scalable vector register, encoded in the "Zm" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(PL) mask = P[g, PL];  
bits(esize) operand1 = V[dn, esize];  
bits(VL) operand2 = Z[m, VL];  
bits(esize) result;  
integer last = LastActiveElement(mask, esize);  
  
if last < 0 then  
    result = ZeroExtend(operand1, esize);  
else  
    if !isBefore then  
        last = last + 1;  
        if last >= elements then last = 0;  
    result = Elem[operand2, last, esize];  
  
V[dn, esize] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

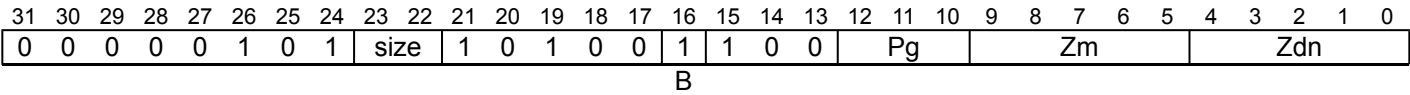
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CLASTB (vectors)

Conditionally extract last element to vector register

From the second source vector register extract the last active element, and then replicate that element to destructively fill the destination and first source vector.

If there are no active elements then leave the destination and source vector unmodified.



CLASTB <Zdn>.<T>, <Pg>, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean isBefore = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
integer last = LastActiveElement(mask, esize);

if last < 0 then
    result = operand1;
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
    for e = 0 to elements-1
        Elem[result, e, esize] = Elem[operand2, last, esize];
Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



CLS

Count leading sign bits (predicated)

Count the number of consecutive sign bits, starting from the most significant bit in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	0	0	0	1	0	1	Pg			Zn				Zd						

```
CLS <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esome) element = Elem[operand, e, esize];
        Elem[result, e, esize] = CountLeadingSignBits(element)<esome-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CLZ

Count leading zero bits (predicated)

Count the number of consecutive binary zero bits, starting from the most significant bit in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	0	0	1	1	0	1	Pg			Zn				Zd						

```
CLZ <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esome) element = Elem[operand, e, esize];
        Elem[result, e, esize] = CountLeadingZeroBits(element)<esome-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CMLA (indexed)

### Complex integer multiply-add with rotate (indexed)

Multiply the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in each 128-bit segment of the first source vector by the specified complex number in the corresponding the second source vector segment rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

Then add the products to the corresponding components of the complex numbers in the addend vector. Destructively place the results in the corresponding elements of the addend vector. This instruction is unpredicated.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

It has encodings from 2 classes: [16-bit](#) and [32-bit](#)

### 16-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	0	1	i2		Zm		0	1	1	0	rot				Zn					Zda			
size<1>size<0>																															

**CMLA** <Zda>.H, <Zn>.H, <Zm>.H[<imm>], <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');
```

### 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	1	1	i1		Zm		0	1	1	0	rot				Zn					Zda			
size<1>size<0>																															

**CMLA** <Zda>.S, <Zn>.S, <Zm>.S[<imm>], <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

- <Zm> For the 16-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 32-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 16-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.  
For the 32-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.
- <const> Is the const specifier, encoded in "rot":

rot	<const>
00	#0
01	#90
10	#180
11	#270

## Operation

```

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
constant integer pairspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for p = 0 to pairs-1
    integer segmentbase = p - (p MOD pairspersegment);
    integer s = segmentbase + index;
    integer elt1_a = SInt(Elem[operand1, 2 * p + sel_a, esize]);
    integer elt2_a = SInt(Elem[operand2, 2 * s + sel_a, esize]);
    integer elt2_b = SInt(Elem[operand2, 2 * s + sel_b, esize]);
    bits(esize) elt3_r = Elem[operand3, 2 * p + 0, esize];
    bits(esize) elt3_i = Elem[operand3, 2 * p + 1, esize];
    integer product_r = elt1_a * elt2_a;
    integer product_i = elt1_a * elt2_b;
    if sub_r then
        Elem[result, 2 * p + 0, esize] = elt3_r - product_r;
    else
        Elem[result, 2 * p + 0, esize] = elt3_r + product_r;
    if sub_i then
        Elem[result, 2 * p + 1, esize] = elt3_i - product_i;
    else
        Elem[result, 2 * p + 1, esize] = elt3_i + product_i;

Z[da, VL] = result;

```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



CMLA (vectors)

Complex integer multiply-add with rotate

Multiply the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in the first source vector by the corresponding complex number in the second source vector rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

Then add the products to the corresponding components of the complex numbers in the addend vector. Destructively place the results in the corresponding elements of the addend vector. This instruction is unpredicated.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		0	Zm					0	0	1	0	rot		Zn					Zda				

CMLA <Zda>.<T>, <Zn>.<T>, <Zm>.<T>, <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in "rot":

rot	<const>
00	#0
01	#90
10	#180
11	#270



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for p = 0 to pairs-1
    integer elt1_a = SInt(Elem[operand1, 2 * p + sel_a, esize]);
    integer elt2_a = SInt(Elem[operand2, 2 * p + sel_a, esize]);
    integer elt2_b = SInt(Elem[operand2, 2 * p + sel_b, esize]);
    bits(esize) elt3_r = Elem[operand3, 2 * p + 0, esize];
    bits(esize) elt3_i = Elem[operand3, 2 * p + 1, esize];
    integer product_r = elt1_a * elt2_a;
    integer product_i = elt1_a * elt2_b;
    if sub_r then
        Elem[result, 2 * p + 0, esize] = elt3_r - product_r;
    else
        Elem[result, 2 * p + 0, esize] = elt3_r + product_r;
    if sub_i then
        Elem[result, 2 * p + 1, esize] = elt3_i - product_i;
    else
        Elem[result, 2 * p + 1, esize] = elt3_i + product_i;

Z[da, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMP<cc> (immediate)

Compare vector to immediate

Compare active integer elements in the source vector with an immediate, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

<cc>	Comparison
EQ	equal
GE	signed greater than or equal
GT	signed greater than
HI	unsigned higher than
HS	unsigned higher than or same
LE	signed less than or equal
LO	unsigned lower than
LS	unsigned lower than or same
LT	signed less than
NE	not equal

It has encodings from 10 classes: [Equal](#) , [Greater than](#) , [Greater than or equal](#) , [Higher](#) , [Higher or same](#) , [Less than](#) , [Less than or equal](#) , [Lower](#) , [Lower or same](#) and [Not equal](#)

Equal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	0	imm5					1	0	0	Pg				Zn				0	Pd				
ne																															

CMPEQ <Pd>.<T>, <Pg>/z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEComp op = Cmp_EQ;
integer imm = Sint(imm5);
boolean unsigned = FALSE;
```

Greater than

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	0	imm5					0	0	0	Pg				Zn				1	Pd				
lt																ne															

CMPGT <Pd>.<T>, <Pg>/z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEComp op = Cmp_GT;
integer imm = Sint(imm5);
boolean unsigned = FALSE;
```

Greater than or equal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	0	imm5					0	0	0	Pg				Zn				0	Pd				
lt																ne															

**CMPGE** <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEComp op = Cmp_GE;
integer imm = SInt(imm5);
boolean unsigned = FALSE;
```

## Higher

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size	1	imm7							0	Pg			Zn				1	Pd					
																		lt										ne			

**CMPHI** <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEComp op = Cmp_GT;
integer imm = UInt(imm7);
boolean unsigned = TRUE;
```

## Higher or same

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size	1	imm7							0	Pg			Zn				0	Pd					
																		lt										ne			

**CMPHS** <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEComp op = Cmp_GE;
integer imm = UInt(imm7);
boolean unsigned = TRUE;
```

## Less than

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	0	imm5							0	0	1	Pg			Zn				0	Pd			
																		lt										ne			

**CMPLT** <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEComp op = Cmp_LT;
integer imm = SInt(imm5);
boolean unsigned = FALSE;
```

## Less than or equal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	0	imm5					0	0	1	Pg			Zn				1	Pd					
																	lt				ne										

**CMPLE** <Pd>.<T>, <Pg>/z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEComp op = Cmp_LE;
integer imm = SInt(imm5);
boolean unsigned = FALSE;
```

## Lower

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size	1	imm7					1	Pg			Zn				0	Pd							
																	lt				ne										

**CMPLO** <Pd>.<T>, <Pg>/z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEComp op = Cmp_LT;
integer imm = UInt(imm7);
boolean unsigned = TRUE;
```

## Lower or same

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size	1	imm7					1	Pg			Zn				1	Pd							
																	lt				ne										

**CMPLS** <Pd>.<T>, <Pg>/z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEComp op = Cmp_LE;
integer imm = UInt(imm7);
boolean unsigned = TRUE;
```

## Not equal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
0	0	1	0	0	1	0	1	size	0	imm5					1	0	0	Pg			Zn				1	Pd													
																												ne											

**CMPNE** <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEComp op = Comp_NE;
integer imm = SInt(imm5);
boolean unsigned = FALSE;
```

### Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <T> Is the size specifier, encoded in "size":
 

size	<T>
00	B
01	H
10	S
11	D
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
- <imm> For the equal, greater than, greater than or equal, less than, less than or equal and not equal variant: is the signed immediate operand, in the range -16 to 15, encoded in the "imm5" field.  
For the higher, higher or same, lower and lower or same variant: is the unsigned immediate operand, in the range 0 to 127, encoded in the "imm7" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  if ActivePredicateElement(mask, e, esize) then
    boolean cond;
    case op of
      when Comp_EQ cond = element1 == imm;
      when Comp_NE cond = element1 != imm;
      when Comp_GE cond = element1 >= imm;
      when Comp_LT cond = element1 < imm;
      when Comp_GT cond = element1 > imm;
      when Comp_LE cond = element1 <= imm;
    bit pbit = if cond then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
  else
    Elem[result, e, psize] = ZeroExtend('0', psize);

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

### Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMP<cc> (vectors)

Compare vectors

Compare active integer elements in the first source vector with corresponding elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

<cc>	Comparison
EQ	equal
GE	signed greater than or equal
GT	signed greater than
HI	unsigned higher than
HS	unsigned higher than or same
NE	not equal

This instruction is used by the pseudo-instructions [CMPLE \(vectors\)](#), [CMPLO \(vectors\)](#), [CMPLS \(vectors\)](#), and [CMPLT \(vectors\)](#).  
It has encodings from 6 classes: [Equal](#) , [Greater than](#) , [Greater than or equal](#) , [Higher](#) , [Higher or same](#) and [Not equal](#)

Equal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size	0	Zm			1	0	1	Pg			Zn			0	Pd								
ne																															

CMPEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_EQ;
boolean unsigned = FALSE;
```

Greater than

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size	0	Zm			1	0	0	Pg		Zn			1	Pd			ne						

CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_GT;
boolean unsigned = FALSE;
```

Greater than or equal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size	0	Zm				1	0	0	Pg			Zn				0	Pd						
ne																															

**CMPGE** <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_GE;
boolean unsigned = FALSE;
```

## Higher

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size	0				Zm		0	0	0		Pg						Zn		1			Pd	

ne

**CMPHI** <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_GT;
boolean unsigned = TRUE;
```

## Higher or same

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size	0				Zm		0	0	0		Pg						Zn		0			Pd	

ne

**CMPHS** <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_GE;
boolean unsigned = TRUE;
```

## Not equal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size	0				Zm		1	0	1		Pg						Zn		1			Pd	

ne

**CMPNE** <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_NE;
boolean unsigned = FALSE;
```



## Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if ActivePredicateElement(mask, e, esize) then
        boolean cond;
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        case op of
            when Cmp_EQ cond = element1 == element2;
            when Cmp_NE cond = element1 != element2;
            when Cmp_GE cond = element1 >= element2;
            when Cmp_LT cond = element1 < element2;
            when Cmp_GT cond = element1 > element2;
            when Cmp_LE cond = element1 <= element2;
        bit pbit = if cond then '1' else '0';
        Elem[result, e, psize] = ZeroExtend(pbit, psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMP<cc> (wide elements)

Compare vector to 64-bit wide elements

Compare active integer elements in the first source vector with overlapping 64-bit doubleword elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

<cc>	Comparison
EQ	equal
GE	signed greater than or equal
GT	signed greater than
HI	unsigned higher than
HS	unsigned higher than or same
LE	signed less than or equal
LO	unsigned lower than
LS	unsigned lower than or same
LT	signed less than
NE	not equal

It has encodings from 10 classes: [Equal](#) , [Greater than](#) , [Greater than or equal](#) , [Higher](#) , [Higher or same](#) , [Less than](#) , [Less than or equal](#) , [Lower](#) , [Lower or same](#) and [Not equal](#)

Equal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size	0			Zm		0	0	1		Pg						Zn		0				Pd	
																ne															

CMPEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Comp_EQ;
boolean unsigned = FALSE;
```

Greater than

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size	0			Zm		0	1	0		Pg						Zn		1				Pd	
																U		lt		ne											

CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Comp_GT;
boolean unsigned = FALSE;
```

Greater than or equal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0									
0	0	1	0	0	1	0	0	size	0	Zm						0	1	0	Pg			Zn					0	Pd												
																U			It																			ne		

CMPGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<D>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_GE;
boolean unsigned = FALSE;
```

Higher

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0											
0	0	1	0	0	1	0	0	size	0	Zm						1	1	0	Pg			Zn					1	Pd														
																U			It																					ne		

CMPHI <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<D>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_GT;
boolean unsigned = TRUE;
```

Higher or same

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0											
0	0	1	0	0	1	0	0	size	0	Zm						1	1	0	Pg			Zn					0	Pd														
																U			It																					ne		

CMPHS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<D>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_GE;
boolean unsigned = TRUE;
```

Less than

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	0	1	0	0	1	0	0	size	0	Zm						0	1	1	Pg			Zn					0	Pd							
																U			It														ne		

**CMPLE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_LT;
boolean unsigned = FALSE;
```

**Less than or equal**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	1	0	0	1	0	0	size	0	Zm						0	1	1	Pg			Zn					1	Pd					
																U		It														ne	

**CMPLT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_LE;
boolean unsigned = FALSE;
```

**Lower**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	0	1	0	0	size	0	Zm						1	1	1	Pg			Zn					0	Pd				
																U		It														ne

**CMPLO <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_LT;
boolean unsigned = TRUE;
```

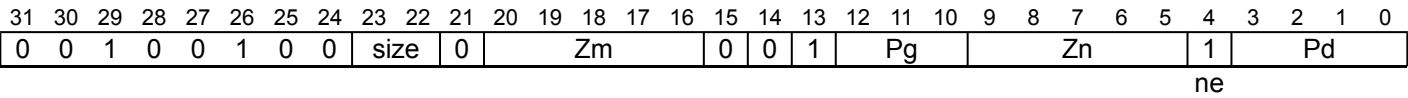
**Lower or same**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	1	0	0	1	0	0	size	0	Zm						1	1	1	Pg			Zn						1	Pd				
																U		It														ne	

CMPLS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_LE;
boolean unsigned = TRUE;
```

Not equal



CMPNE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_NE;
boolean unsigned = FALSE;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	RESERVED

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if ActivePredicateElement(mask, e, esize) then
        boolean cond;
        integer element2 = Int(Elem[operand2, (e * esize) DIV 64, 64], unsigned);
        case op of
            when Cmp_EQ cond = element1 == element2;
            when Cmp_NE cond = element1 != element2;
            when Cmp_GE cond = element1 >= element2;
            when Cmp_LT cond = element1 < element2;
            when Cmp_GT cond = element1 > element2;
            when Cmp_LE cond = element1 <= element2;
        bit pbit = if cond then '1' else '0';
        Elem[result, e, psize] = ZeroExtend(pbit, psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMPLE (vectors)

Compare signed less than or equal to vector, setting the condition flags

Compare active signed integer elements in the first source vector being less than or equal to corresponding signed elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of [CMP<cc> \(vectors\)](#). This means:

- The encodings in this description are named to match the encodings of [CMP<cc> \(vectors\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [CMP<cc> \(vectors\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size		0	Zm				1	0	0	Pg			Zn				0		Pd				
ne																															

CMPLE <Pd>.<T>, <Pg>/z, <Zm>.<T>, <Zn>.<T>

is equivalent to

CMPGE <Pd>.<T>, <Pg>/z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of [CMP<cc> \(vectors\)](#) gives the operational pseudocode for this instruction.

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.

CMPLO (vectors)

Compare unsigned lower than vector, setting the condition flags

Compare active unsigned integer elements in the first source vector being lower than corresponding unsigned elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of [CMP<cc> \(vectors\)](#). This means:

- The encodings in this description are named to match the encodings of [CMP<cc> \(vectors\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [CMP<cc> \(vectors\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size		0	Zm				0	0	0	Pg			Zn				1		Pd				
ne																															

CMPLO <Pd>.<T>, <Pg>/z, <Zm>.<T>, <Zn>.<T>

is equivalent to

CMPHI <Pd>.<T>, <Pg>/z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

- <Pd>

Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

- <Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of [CMP<cc> \(vectors\)](#) gives the operational pseudocode for this instruction.

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.



CMPLS (vectors)

Compare unsigned lower or same as vector, setting the condition flags

Compare active unsigned integer elements in the first source vector being lower than or same as corresponding unsigned elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of [CMP<cc> \(vectors\)](#). This means:

- The encodings in this description are named to match the encodings of [CMP<cc> \(vectors\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [CMP<cc> \(vectors\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size		0	Zm				0	0	0	Pg			Zn				0	Pd					
ne																															

CMPLS <Pd>.<T>, <Pg>/z, <Zm>.<T>, <Zn>.<T>

is equivalent to

CMPHS <Pd>.<T>, <Pg>/z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of [CMP<cc> \(vectors\)](#) gives the operational pseudocode for this instruction.

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.

CMPLT (vectors)

Compare signed less than vector, setting the condition flags

Compare active signed integer elements in the first source vector being less than corresponding signed elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of [CMP<cc> \(vectors\)](#). This means:

- The encodings in this description are named to match the encodings of [CMP<cc> \(vectors\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [CMP<cc> \(vectors\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size		0	Zm				1	0	0	Pg				Zn				1	Pd				
ne																															

CMPLT <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

- <Pd>

Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

- <Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of [CMP<cc> \(vectors\)](#) gives the operational pseudocode for this instruction.

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.

CNOT

Logically invert boolean condition in vector (predicated)

Logically invert the boolean value in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

Boolean TRUE is any non-zero value in a source, and one in a result element. Boolean FALSE is always zero.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	0	1	1	1	0	1	Pg			Zn				Zd						

```
CNOT <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = ZeroExtend(IsZeroBit(element), esize);

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CNT

Count non-zero bits (predicated)

Count non-zero bits in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	0	1	0	1	0	0	1	Pg						Zn						Zd

CNT <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = BitCount(element)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## CNTB, CNTD, CNTH, CNTW

Set scalar to multiple of predicate constraint element count

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then places the result in the scalar destination.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 4 classes: [Byte](#) , [Doubleword](#) , [Halfword](#) and [Word](#)

### Byte

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	1	0	imm4				1	1	1	0	0	0	pattern					Rd				

size<1>size<0>

CNTB <Xd>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

### Doubleword

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	1	1	0	imm4				1	1	1	0	0	0	pattern					Rd				

size<1>size<0>

CNTD <Xd>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

### Halfword

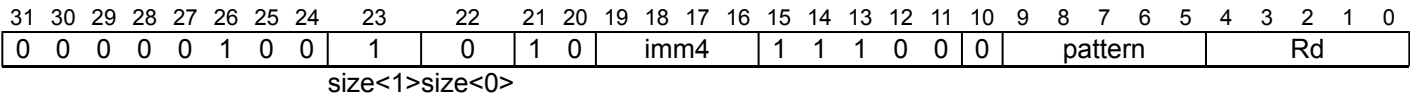
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	1	1	0	imm4				1	1	1	0	0	0	pattern					Rd				

size<1>size<0>

CNTH <Xd>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

Word



CNTW <Xd>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

Assembler Symbols

- <Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.
- <pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

- <imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
X[d, 64] = (count * imm)<63:0>;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.





CNTP (predicate as counter)

Set scalar to count from predicate-as-counter

Counts the number of true elements in the source predicate and places the scalar result in the destination general-purpose register.

SVE2  
(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	0	0	0	0	0	1	0	0	0	0	vl	1	PNn				Rd					

CNTP <Xd>, <PNn>.<T>, <vl>

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(PNn);
integer d = UInt(Rd);
constant integer width = 2 << UInt(vl);
```

Assembler Symbols

- <Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.
- <PNn> Is the name of the first source scalable predicate register, with predicate-as-counter encoding, encoded in the "PNn" field.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

- <vl> Is the vl specifier, encoded in "vl":

vl	<vl>
0	VLx2
1	VLx4

Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) pred = P[n, PL];
bits(PL*4) mask = CounterToPredicate(pred<15:0>, PL*4);
bits(64) sum = Zeros(64);
constant integer limit = elements * width;

for e = 0 to limit-1
  if ActivePredicateElement(mask, e, esize) then
    sum = sum + 1;
X[d, 64] = sum;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CNTP (predicate)

Set scalar to count of true predicate elements

Counts the number of active and true elements in the source predicate and places the scalar result in the destination general-purpose register. Inactive predicate elements are not counted.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	0	0	0	0	0	1	0			Pg		0			Pn					Rd		

CNTP <Xd>, <Pg>, <Pn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Rd);
```

Assembler Symbols

- <Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand = P[n, PL];
bits(64) sum = Zeros(64);

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) && ActivePredicateElement(operand, e, esize) then
        sum = sum + 1;
X[d, 64] = sum;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.



COMPACT

Shuffle active elements of vector to the right and fill with zero

Read the active elements from the source vector and pack them into the lowest-numbered elements of the destination vector. Then set any remaining elements of the destination vector to zero.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	0	0	0	0	1	1	0	0	Pg				Zn				Zd					

COMPACT <Zd>.<T>, <Pg>, <Zn>.<T>

```
if !HaveSVE() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size<0>":

size<0>	<T>
0	S
1	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Zeros(VL);
integer x = 0;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand1, e, esize];
        Elem[result, x, esize] = element;
        x = x + 1;

Z[d, VL] = result;
```

CPY (immediate, merging)

Copy signed integer immediate to vector elements (merging)

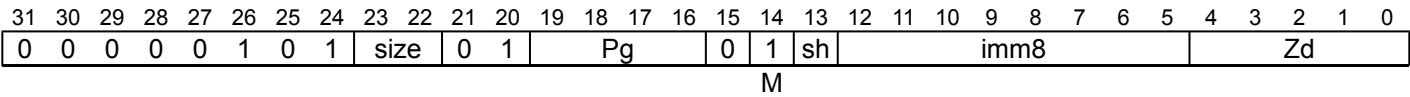
Copy a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<simm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This instruction is used by the alias [MOV \(immediate, predicated, merging\)](#).

This instruction is used by the pseudo-instruction [FMOV \(zero, predicated\)](#).



CPY <Zd>.<T>, <Pg>/M, #<imm>{, <shift>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Zd);
boolean merging = TRUE;
integer imm = SInt(imm8);
if sh == '1' then imm = imm << 8;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<imm> Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #8

Alias Conditions

Alias	Is preferred when
<a href="#">FMOV (zero, predicated)</a>	Never
<a href="#">MOV (immediate, predicated, merging)</a>	Unconditionally

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) dest = Z[d, VL];
bits(VL) result;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = imm<esize-1:0>;
    elsif merging then
        Elem[result, e, esize] = Elem[dest, e, esize];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



CPY (immediate, zeroing)

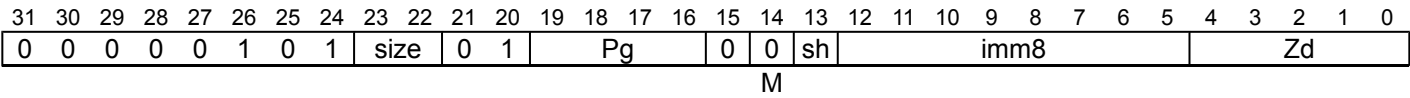
Copy signed integer immediate to vector elements (zeroing)

Copy a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register are set to zero.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This instruction is used by the alias [MOV\(immediate, predicated, zeroing\)](#).



CPY <Zd>.<T>, <Pg>/z, #<imm>{, <shift>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Zd);
boolean merging = FALSE;
integer imm = SInt(imm8);
if sh == '1' then imm = imm << 8;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<imm> Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #8

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) dest = Z[d, VL];
bits(VL) result;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = imm<esize-1:0>;
    elsif merging then
        Elem[result, e, esize] = Elem[dest, e, esize];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CPY (scalar)

Copy general-purpose register to vector elements (predicated)

Copy the general-purpose scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is used by the alias [MOV \(scalar, predicated\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size		1	0	1	0	0	0	1	0	1	Pg			Rn			Zd						

CPY <Zd>.<T>, <Pg>/M, <R><n|SP>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Rn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<R> Is a width specifier, encoded in "size":

size	<R>
01	W
x0	W
11	X

<n|SP> Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) result = Z[d, VL];
if AnyActiveElement(mask, esize) then
    bits(64) operand1;
    if n == 31 then
        operand1 = SP[];
    else
        operand1 = X[n, 64];

    for e = 0 to elements-1
        if ActivePredicateElement(mask, e, esize) then
            Elem[result, e, esize] = operand1<esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CPY (SIMD&FP scalar)

Copy SIMD&FP scalar register to vector elements (predicated)

Copy the SIMD & floating-point scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is used by the alias [MOV \(SIMD&FP scalar, predicated\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size		1	0	0	0	0	0	1	0	0	Pg			Vn			Zd						

CPY <Zd>.<T>, <Pg>/M, <V><n>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Vn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<V> Is a width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	D

<n> Is the number [0-31] of the source SIMD&FP register, encoded in the "Vn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(esize) operand1 = if AnyActiveElement(mask, esize) then V[n, esize] else Zeros(esize);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = operand1;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CTERMEQ, CTERMNE

Compare and terminate loop

Detect termination conditions in serialized vector loops. Tests whether the comparison between the scalar source operands holds true and if not tests the state of the !LAST condition flag (C) which indicates whether the previous flag-setting predicate instruction selected the last element of the vector partition.  
The Z and C condition flags are preserved by this instruction. The N and V condition flags are set as a pair to generate one of the following conditions for a subsequent conditional instruction:

Condition	N	V	Meaning
GE	0	0	Continue loop (compare failed and last element not selected)
LT	0	1	Terminate loop (last element selected)
LT	1	0	Terminate loop (compare succeeded)
GE	1	1	Never generated

The scalar source operands are 32-bit or 64-bit general-purpose registers of the same size.  
It has encodings from 2 classes: [Equal](#) and [Not equal](#)

Equal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	1	sz	1	Rm					0	0	1	0	0	0	Rn					0	0	0	0	0
ne																															

CTERMEQ <R><n>, <R><m>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Rn);
integer m = UInt(Rm);
SVEComp op = Cmp_EQ;
```

Not equal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	1	sz	1	Rm					0	0	1	0	0	0	Rn					1	0	0	0	0
ne																															

CTERMNE <R><n>, <R><m>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Rn);
integer m = UInt(Rm);
SVEComp op = Cmp_NE;
```

Assembler Symbols

- <R>

Is a width specifier, encoded in “sz”:

sz	<R>
0	W
1	X
- <n>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.
- <m>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
bits(esize) operand1 = X[n, esize];
bits(esize) operand2 = X[m, esize];
integer element1 = UInt(operand1);
integer element2 = UInt(operand2);
boolean term;

case op of
  when Cmp_EQ term = element1 == element2;
  when Cmp_NE term = element1 != element2;
if term then
  PSTATE.N = '1';
  PSTATE.V = '0';
else
  PSTATE.N = '0';
  PSTATE.V = (NOT PSTATE.C);
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## DECB, DECD, DECH, DECW (scalar)

Decrement scalar by multiple of predicate constraint element count

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 4 classes: [Byte](#) , [Doubleword](#) , [Halfword](#) and [Word](#)

### Byte

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	1	1	imm4				1	1	1	0	0	1	pattern					Rdn				
size<1>size<0>																D															

DECB <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

### Doubleword

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
0	0	0	0	0	1	0	0	1	1	1	1	imm4				1	1	1	0	0	1	pattern					Rdn										
size<1>size<0>																					D																

DECD <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

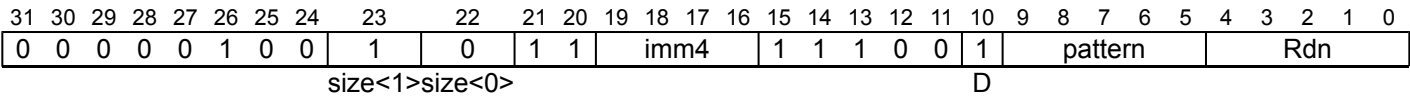
### Halfword

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	1	1	1	imm4				1	1	1	0	0	1	pattern					Rdn				
size<1>size<0>																D															

DECH <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

Word



```
DECW <Xdn>{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

Assembler Symbols

- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

- <imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
constant integer VL = CurrentVL;
bits(64) operand1 = X[dn, 64];

X[dn, 64] = operand1 - (count * imm);
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



## DECD, DECH, DECW (vector)

Decrement vector by multiple of predicate constraint element count

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 3 classes: [Doubleword](#) , [Halfword](#) and [Word](#)

### Doubleword

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
0	0	0	0	0	1	0	0	1	1	1	1	imm4				1	1	0	0	0	1	pattern					Zdn										
size<1>size<0>																					D																

```
DECD <Zdn>.D{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

### Halfword

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	1	0	0	0	1	1	1	imm4				1	1	0	0	0	1	pattern					Zdn					
size<1>size<0>																					D											

```
DECH <Zdn>.H{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

### Word

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
0	0	0	0	0	1	0	0	1	0	1	1	imm4				1	1	0	0	0	1	pattern					Zdn										
size<1>size<0>																					D																

```
DECW <Zdn>.S{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

### Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    Elem[result, e, esize] = Elem[operand1, e, esize] - (count * imm);

Z[dn, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

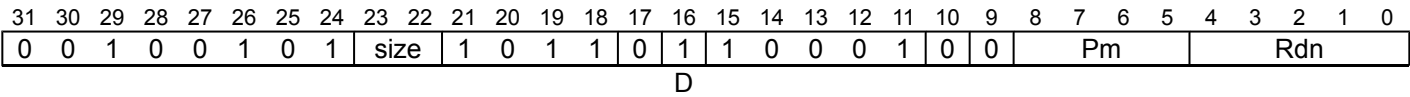
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

DECP (scalar)

Decrement scalar by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to decrement the scalar destination.



DECP <Xdn>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
```

Assembler Symbols

- <Xdn>Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Pm>Is the name of the source scalable predicate register, encoded in the "Pm" field.
- <T>Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) operand1 = X[dn, 64];
bits(PL) operand2 = P[m, PL];
integer count = 0;

for e = 0 to elements-1
    if ActivePredicateElement(operand2, e, esize) then
        count = count + 1;

X[dn, 64] = operand1 - count;
```

Operational information

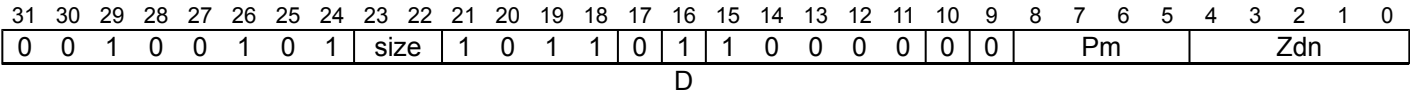
If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

DECP (vector)

Decrement vector by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to decrement all destination vector elements.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.



```
DECP <Zdn>.<T>, <Pm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) operand2 = P[m, PL];
bits(VL) result;
integer count = 0;

for e = 0 to elements-1
    if ActivePredicateElement(operand2, e, esize) then
        count = count + 1;

for e = 0 to elements-1
    Elem[result, e, esize] = Elem[operand1, e, esize] - count;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



DUP (immediate)

Broadcast signed immediate to vector elements (unpredicated)

Unconditionally broadcast the signed integer immediate into each element of the destination vector. This instruction is unpredicated.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This instruction is used by the alias [MOV \(immediate, unpredicated\)](#).

This instruction is used by the pseudo-instruction [FMOV \(zero, unpredicated\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	1	1	0	0	0	1	1	sh	imm8								Zd					

DUP <Zd>.<T>, #<imm>{, <shift>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer d = UInt(Zd);
integer imm = SInt(imm8);
if sh == '1' then imm = imm << 8;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm> Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #8

Alias Conditions

Alias	Is preferred when
<a href="#">FMOV (zero, unpredicated)</a>	Never
<a href="#">MOV (immediate, unpredicated)</a>	Unconditionally

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(VL) result = Replicate(imm<esize-1:0>, VL DIV esize);
Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DUP (indexed)

Broadcast indexed element to vector (unpredicated)

Unconditionally broadcast the indexed source vector element into each element of the destination vector. This instruction is unpredicated.  
The immediate element index is in the range of 0 to 63 (bytes), 31 (halfwords), 15 (words), 7 (doublewords) or 3 (quadwords). Selecting an element beyond the accessible vector length causes the destination vector to be set to zero.  
This instruction is used by the alias [MOV \(SIMD&FP scalar, unpredicated\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	imm2	1				tsz			0	0	1	0	0	0				Zn				Zd		

DUP <Zd>.<T>, <Zn>.<T>[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if tsz == '00000' then UNDEFINED;
constant integer lsb = LowestSetBit(tsz);
constant integer esize = 8 << lsb;
constant bits(7) imm = imm2:tsz;
constant integer index = UInt(imm<6:(lsb+1)>);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <T> Is the size specifier, encoded in "tsz":

tsz	<T>
00000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D
10000	Q
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
- <imm> Is the immediate index, in the range 0 to one less than the number of elements in 512 bits, encoded in "imm2:tsz".

Alias Conditions

Alias	Is preferred when
<a href="#">MOV (SIMD&amp;FP scalar, unpredicated)</a>	<a href="#">BitCount</a> (imm2:tsz) == 1
<a href="#">MOV (SIMD&amp;FP scalar, unpredicated)</a>	<a href="#">BitCount</a> (imm2:tsz) > 1

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(VL) operand1 = Z[n, VL];  
bits(VL) result;  
bits(esize) element;  
  
if index >= elements then  
    element = Zeros(esize);  
else  
    element = Elem[operand1, index, esize];  
result = Repliate(element, VL DIV esize);  
  
Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DUP (scalar)

Broadcast general-purpose register to vector elements (unpredicated)

Unconditionally broadcast the general-purpose scalar source register into each element of the destination vector. This instruction is unpredicated.

This instruction is used by the alias [MOV \(scalar, unpredicated\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size		1	0	0	0	0	0	0	0	1	1	1	0	Rn					Zd				

DUP <Zd>.<T>, <R><n|SP>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in "size":

size	<R>
01	W
x0	W
11	X

<n|SP> Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer PL = VL DIV 8;
bits(64) operand;
if n == 31 then
    operand = SP[];
else
    operand = X[n, 64];
bits(VL) result;

for e = 0 to elements-1
    Elem[result, e, esize] = operand<esize-1:0>;

Z[d, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DUPM

Broadcast logical bitmask immediate to vector (unpredicated)

Unconditionally broadcast the logical bitmask immediate into each element of the destination vector. This instruction is unpredicated. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits.

This instruction is used by the alias [MOV](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	1	1	0	0	0	0	imm13														Zd			

DUPM <Zd>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer d = UInt(Zd);
bits(esize) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE, esize);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "imm13<12>:imm13<5:0>":

imm13<12>	imm13<5:0>	<T>
0	0xxxxx	S
0	10xxxx	H
0	110xxx	B
0	1110xx	B
0	11110x	B
0	111110	RESERVED
0	111111	RESERVED
1	xxxxxx	D

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

Alias Conditions

Alias	Is preferred when
<a href="#">MOV</a>	<a href="#">SVEMoveMaskPreferred</a> (imm13)

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) result = Replicate(imm, VL DIV esize);
Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.





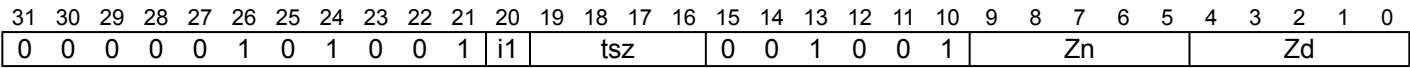
DUPQ

Broadcast indexed element within each quadword vector segment (unpredicated)

Unconditionally broadcast the indexed element within each 128-bit source vector segment to all elements of the corresponding destination vector segment. This instruction is unpredicated.

The immediate element index is in the range of 0 to 15 (bytes), 7 (halfwords), 3 (words) or 1 (doublewords).

SVE2  
(FEAT\_SVE2p1)



DUPQ <Zd>.<T>, <Zn>.<T>[<imm>]

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
if tsz == '0000' then UNDEFINED;
constant integer lsb = LowestSetBit(tsz);
constant integer esize = 8 << lsb;
constant bits(5) imm = i1:tsz;
constant integer index = UInt(imm<4:(lsb+1)>);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tsz":

tsz	<T>
0000	RESERVED
xxx1	B
xx10	H
x100	S
1000	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<imm> Is the immediate index, in the range 0 to one less than the number of elements in 128 bits, encoded in "i1:tsz".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elements = 128 DIV esize;
bits(VL) operand = Z[n, VL];
bits(VL) result;
bits(esize) element;

for s = 0 to segments-1
    element = Elem[operand, s * elements + index, esize];
    Elem[result, s, 128] = Replicate(element, 128 DIV esize);

Z[d, VL] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EON

Bitwise exclusive OR with inverted immediate (unpredicated)

Bitwise exclusive OR an inverted immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This is a pseudo-instruction of [EOR \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [EOR \(immediate\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [EOR \(immediate\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	1	0	0	0	0	imm13														Zdn			

EON <Zdn>.<T>, <Zdn>.<T>, #<const>

is equivalent to

EOR <Zdn>.<T>, <Zdn>.<T>, #(-<const> - 1)

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "imm13<12>:imm13<5:0>":

imm13<12>	imm13<5:0>	<T>
0	0xxxxx	S
0	10xxxx	H
0	110xxx	B
0	1110xx	B
0	11110x	B
0	111110	RESERVED
0	111111	RESERVED
1	xxxxxx	D

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

Operation

The description of [EOR \(immediate\)](#) gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



EOR (immediate)

Bitwise exclusive OR with immediate (unpredicated)

Bitwise exclusive OR an immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction [EON](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	1	0	0	0	0	imm13														Zdn			

```
EOR <Zdn>.<T>, <Zdn>.<T>, #<const>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer dn = UInt(Zdn);
bits(64) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE, 64);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "imm13<12>:imm13<5:0>":

imm13<12>	imm13<5:0>	<T>
0	0xxxxxx	S
0	10xxxxx	H
0	110xxxx	B
0	1110xxx	B
0	11110xx	B
0	1111110	RESERVED
0	1111111	RESERVED
1	xxxxxxx	D

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 64;
bits(VL) operand = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(64) element1 = Elem[operand, e, 64];
    Elem[result, e, 64] = element1 EOR imm;

Z[dn, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EOR (predicates)

Bitwise exclusive OR predicates

Bitwise exclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This instruction is used by the alias [NOT \(predicate\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	0	0	0	Pm				0	1	Pg				1	Pn				0	Pd			
S																															

EOR <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
<a href="#">NOT (predicate)</a>	Pm == Pg

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit element1 = PredicateElement(operand1, e, esize);
    bit element2 = PredicateElement(operand2, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, psize] = ZeroExtend(element1 EOR element2, psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



EOR (vectors, predicated)

Bitwise exclusive OR vectors (predicated)

Bitwise exclusive OR active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	0	0	1	0	0	0	Pg	Zm				Zdn								

EOR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  if ActivePredicateElement(mask, e, esize) then
    Elem[result, e, esize] = element1 EOR element2;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# EOR (vectors, unpredicated)

Bitwise exclusive OR vectors (unpredicated)

Bitwise exclusive OR all elements of the second source vector with corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	0	1	Zm					0	0	1	1	0	0	Zn					Zd				

EOR <Zd>.D, <Zn>.D, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

## Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];

Z[d, VL] = operand1 EOR operand2;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EOR3

Bitwise exclusive OR of three vectors

Bitwise exclusive OR the corresponding elements of all three source vectors, and destructively place the results in the corresponding elements of the destination and first source vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	1	Zm					0	0	1	1	1	0	Zk					Zdn				

```
EOR3 <Zdn>.D, <Zdn>.D, <Zm>.D, <Zk>.D
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);
```

Assembler Symbols

- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <Zk> Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[k, VL];

Z[dn, VL] = operand1 EOR operand2 EOR operand3;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

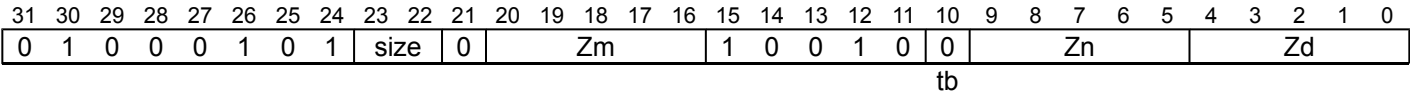
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

EORBT

Interleaving exclusive OR (bottom, top)

Interleaving exclusive OR between the even-numbered elements of the first source vector register and the odd-numbered elements of the second source vector register, placing the result in the even-numbered elements of the destination vector, leaving the odd-numbered elements unchanged. This instruction is unpredicated.



EORBT <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 1;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, 2*e + sel1, esize];
    bits(esize) element2 = Elem[operand2, 2*e + sel2, esize];
    Elem[result, 2*e + sel1, esize] = element1 EOR element2;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# EORQV

Bitwise exclusive OR reduction of quadword vector segments

Bitwise exclusive OR of the same element numbers from each 128-bit source vector segment, placing each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as all zeros.

## SVE2 (FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	1	0	1	0	0	1	Pg			Zn				Vd						

EORQV <Vd>.<T>, <Pg>, <Zn>.<Tb>

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

### Assembler Symbols

<Vd> Is the name of the destination SIMD&FP register, encoded in the "Vd" field.

<T> Is an arrangement specifier, encoded in "size":

size	<T>
00	16B
01	8H
10	4S
11	2D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	B
01	H
10	S
11	D

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elemperssegment = 128 DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(128) result = Zeros(128);
bits(128) stmp = Zeros(128);

bits(esize) dtmp;

for e = 0 to elemperssegment-1
    dtmp = Zeros(esize);
    for s = 0 to segments-1
        if ActivePredicateElement(mask, s * elemperssegment + e, esize) then
            stmp = Elem[operand, s, 128];
            dtmp = dtmp EOR Elem[stmp, e, esize];
        Elem[result, e, esize] = dtmp<esize-1:0>;

V[d, 128] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

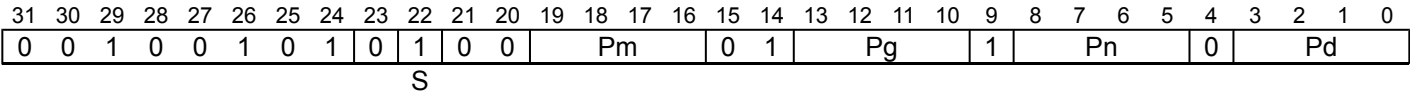


EORS

Bitwise exclusive OR predicates, setting the condition flags

Bitwise exclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the alias [NOTS](#).



```
EORS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
<a href="#">NOTS</a>	Pm == Pg

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit element1 = PredicateElement(operand1, e, esize);
    bit element2 = PredicateElement(operand2, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, psize] = ZeroExtend(element1 EOR element2, psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

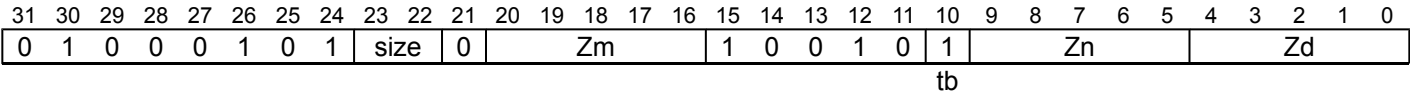
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EORTB

Interleaving exclusive OR (top, bottom)

Interleaving exclusive OR between the odd-numbered elements of the first source vector register and the even-numbered elements of the second source vector register, placing the result in the odd-numbered elements of the destination vector, leaving the even-numbered elements unchanged. This instruction is unpredicated.



EORTB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 0;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, 2*e + sel1, esize];
    bits(esize) element2 = Elem[operand2, 2*e + sel2, esize];
    Elem[result, 2*e + sel1, esize] = element1 EOR element2;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EORV

Bitwise exclusive OR reduction to scalar

Bitwise exclusive OR horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as zero.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	0	0	1	0	0	1	Pg													

```
EORV <V><d>, <Pg>, <Zn>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) result = Zeros(esize);

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        result = result EOR Elem[operand, e, esize];

V[d, esize] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EXT

Extract vector from pair of vectors

Copy the indexed byte up to the last byte of the first source vector to the bottom of the result vector, then fill the remainder of the result starting from the first byte of the second source vector. The result is placed destructively in the destination and first source vector, or constructively in the destination vector. This instruction is unpredicated. An index that is greater than or equal to the vector length in bytes is treated as zero, resulting in the first source vector being copied to the result unchanged.

The Destructive encoding of this instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE: The MOVPRFX instruction must be unpredicated. The MOVPRFX instruction must specify the same destination register as this instruction. The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

It has encodings from 2 classes: [Constructive](#) and [Destructive](#)

Constructive

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	1	1	imm8h				0	0	0	imm8l				Zn				Zd					

EXT <Zd>.B, { <Zn1>.B, <Zn2>.B }, #<imm>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dst = UInt(Zd);
integer s1 = UInt(Zn);
integer s2 = (s1 + 1) MOD 32;
constant integer position = UInt(imm8h:imm8l) * 8;
```

Destructive

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	0	1	imm8h				0	0	0	imm8l				Zm				Zdn					

EXT <Zdn>.B, <Zdn>.B, <Zm>.B, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dst = UInt(Zdn);
integer s1 = dst;
integer s2 = UInt(Zm);
constant integer position = UInt(imm8h:imm8l) * 8;
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded in the "Zn" field.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded in the "Zn" field.
- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <imm> Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8h:imm8l" fields.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[s1, VL];
bits(VL) operand2 = Z[s2, VL];
bits(VL) result;

bits(VL*2) concat = operand2 : operand1;

if position >= VL then
    result = concat<VL-1:0>;
else
    result = concat<(position+VL)-1:position>;

Z[dst, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# EXTQ

Extract vector segment from each pair of quadword vector segments

For each 128-bit vector segment of the result, copy the indexed byte up to and including the last byte of the corresponding first source vector segment to the bottom of the result segment, then fill the remainder of the result segment starting from the first byte of the corresponding second source vector segment. The result segments are destructively placed in the corresponding first source vector segment. This instruction is unpredicated.

## SVE2 (FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	1	1	0	imm4				0	0	1	0	0	1	Zm				Zdn					

EXTQ <Zdn>.B, <Zdn>.B, <Zm>.B, #<imm>

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
integer dn = UInt(Zdn);
integer m = UInt(Zm);
constant integer position = UInt(imm4) << 3;
```

### Assembler Symbols

- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <imm> Is the unsigned immediate operand, in the range 0 to 15, encoded in the "imm4" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for s = 0 to segments-1
    bits(256) concat = Elem[operand2, s, 128] : Elem[operand1, s, 128];
    Elem[result, s, 128] = concat<position+127:position>;

Z[dn, VL] = result;
```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



FABD

Floating-point absolute difference (predicated)

Compute the absolute difference of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the result in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	1	0	0	0	1	0	0	Pg	Zm				Zdn								

FABD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ActivePredicateElement(mask, e, esize) then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAbs(FPSub(element1, element2, FPCR[]));
  else
    Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FABS

Floating-point absolute value (predicated)

Take the absolute value of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This clears the sign bit and cannot signal a floating-point exception. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	1	0	0	1	0	1	Pg	Zn				Zd								

FABS <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPAbs(element);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



FAC<cc>

Floating-point absolute compare vectors

Compare active absolute values of floating-point elements in the first source vector with corresponding absolute values of elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

<cc>	Comparison
GE	greater than or equal
GT	greater than
LE	less than or equal
LT	less than

This instruction is used by the pseudo-instructions [FACLE](#), and [FACLT](#).  
It has encodings from 2 classes: [Greater than](#) and [Greater than or equal](#)

Greater than

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	Zm						1	1	1	Pg			Zn					1	Pd			

FACGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_GT;
```

Greater than or equal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	Zm			1	1	0	Pg			Zn			1	Pd								

FACGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_GE;
```

Assembler Symbols

- <Pd>

Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <T>

Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D
- <Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        boolean res;
        case op of
            when Cmp\_GE res = FPCompareGE(FPAbs(element1), FPAbs(element2), FPCR[]);
            when Cmp\_GT res = FPCompareGT(FPAbs(element1), FPAbs(element2), FPCR[]);
        bit pbit = if res then '1' else '0';
        Elem[result, e, psize] = ZeroExtend(pbit, psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

P[d, PL] = result;

```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



FACLE

Floating-point absolute compare less than or equal

Compare active absolute values of floating-point elements in the first source vector being less than or equal to corresponding absolute values of elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of [FAC<cc>](#). This means:

- The encodings in this description are named to match the encodings of [FAC<cc>](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [FAC<cc>](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size			0	Zm				1	1	0	Pg			Zn				1	Pd				

**FACLE** <Pd>.<T>, <Pg>/z, <Zm>.<T>, <Zn>.<T>

is equivalent to

**FACGE** <Pd>.<T>, <Pg>/z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of [FAC<cc>](#) gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

FACLT

Floating-point absolute compare less than

Compare active absolute values of floating-point elements in the first source vector being less than corresponding absolute values of elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of [FAC<cc>](#). This means:

- The encodings in this description are named to match the encodings of [FAC<cc>](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [FAC<cc>](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size			0	Zm				1	1	1	Pg			Zn				1	Pd				

**FACLT** <Pd>.<T>, <Pg>/z, <Zm>.<T>, <Zn>.<T>

is equivalent to

**FACGT** <Pd>.<T>, <Pg>/z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of [FAC<cc>](#) gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FADD (immediate)

Floating-point add immediate (predicated)

Add an immediate to each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	1	1	0	0	0	1	0	0	Pg	0	0	0	0	i1	Zdn							

FADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esome) imm = if i1 == '0' then FPPointFive('0', esize) else FPOne('0', esize);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in "i1":

i1	<const>
0	#0.5
1	#1.0

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ActivePredicateElement(mask, e, esize) then
    Elem[result, e, esize] = FAdd(element1, imm, FPCR[]);
  else
    Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FADD (vectors, predicated)

Floating-point add vector (predicated)

Add active floating-point elements of the second source vector to corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	!= 00	0	0	0	0	0	0	0	1	0	0	Pg				Zm				Zdn				
size																															

**FADD** <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

### Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

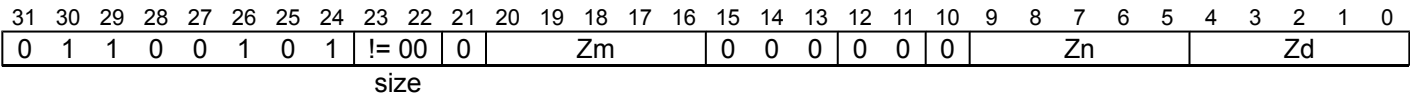
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FADD (vectors, unpredicated)

Floating-point add vector (unpredicated)

Add all floating-point elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.



```
FADD <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);

Z[d, VL] = result;
```

FADDA

Floating-point add strictly-ordered reduction, accumulating in scalar

Floating-point add a SIMD&FP scalar source and all active lanes of the vector source and place the result destructively in the SIMD&FP scalar source register. Vector elements are processed strictly in order from low to high, with the scalar source providing the initial value. Inactive elements in the source vector are ignored. This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	1	1	0	0	0	0	0	1	Pg			Zm			Vdn						

FADDA <V><dn>, <Pg>, <V><dn>, <Zm>.<T>

```
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Vdn);
integer m = UInt(Zm);
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	D

<dn> Is the number [0-31] of the source and destination SIMD&FP register, encoded in the "Vdn" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the source scalable vector register, encoded in the "Zm" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(esize) operand1 = V[dn, esize];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(esize) result = operand1;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand2, e, esize];
        result = FPAdd(result, element, FPCR[]);

V[dn, esize] = result;
```





FADDP

Floating-point add pairwise

Add pairs of adjacent floating-point elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	size	0	1	0	0	0	0	1	0	0	Pg	Zm				Zdn								

```
FADDP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result = Z[dn, VL];
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    if IsEven(e) then
      element1 = Elem[operand1, e + 0, esize];
      element2 = Elem[operand1, e + 1, esize];
    else
      element1 = Elem[operand2, e - 1, esize];
      element2 = Elem[operand2, e + 0, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

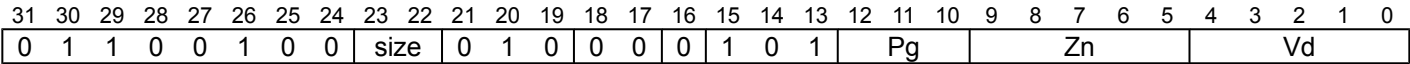
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FADDQV

Floating-point add recursive reduction of quadword vector segments

Floating-point addition of the same element numbers from each 128-bit source vector segment using a recursive pairwise reduction, placing each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as +0.0.

SVE2
(FEAT\_SVE2p1)



FADDQV <Vd>.<T>, <Pg>, <Zn>.<Tb>

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<Vd> Is the name of the destination SIMD&FP register, encoded in the "Vd" field.

<T> Is an arrangement specifier, encoded in "size":

size	<T>
00	RESERVED
01	8H
10	4S
11	2D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	H
10	S
11	D

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elemperssegment = 128 DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPZero('0', esize);
bits(128) result = Zeros(128);

constant integer p2bits = CeilPow2(segments*esize);
constant integer p2elems = p2bits DIV esize;

for e = 0 to elemperssegment-1
    bits(p2bits) stmp;
    bits(esize) dtmp;
    for s = 0 to p2elems-1
        if s < segments && ActivePredicateElement(mask, s * elemperssegment + e, esize) then
            Elem[stmp, s, esize] = Elem[operand, s * elemperssegment + e, esize];
        else
            Elem[stmp, s, esize] = identity;
    dtmp = Reduce(ReduceOp\_FADD, stmp, esize);
    Elem[result, e, esize] = dtmp;
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FADDV

Floating-point add recursive reduction to scalar

Floating-point add horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as +0.0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	0	0	0	0	0	0	0	1	Pg			Zn				Vd					

FADDV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPZero('0', esize);

V[d, esize] = ReducePredicated(ReduceOp_FADD, operand, mask, identity);
```

FCADD

Floating-point complex add with rotate (predicated)

Add the real and imaginary components of the active floating-point complex numbers from the first source vector to the complex numbers from the second source vector which have first been rotated by 90 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, equivalent to multiplying the complex numbers in the second source vector by  $\pm j$  beforehand. Destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	size			0	0	0	0	0	rot	1	0	0	Pg			Zm			Zdn					

FCADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean sub_i = (rot == '0');
boolean sub_r = (rot == '1');
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in "rot":

rot	<const>
0	#90
1	#270

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for p = 0 to pairs-1
    acc_r = Elem[operand1, 2 * p + 0, esize];
    acc_i = Elem[operand1, 2 * p + 1, esize];
    if ActivePredicateElement(mask, 2 * p + 0, esize) then
        elt2_i = Elem[operand2, 2 * p + 1, esize];
        if sub_i then elt2_i = FPNeg(elt2_i);
        acc_r = FPAdd(acc_r, elt2_i, FPCR[]);
    if ActivePredicateElement(mask, 2 * p + 1, esize) then
        elt2_r = Elem[operand2, 2 * p + 0, esize];
        if sub_r then elt2_r = FPNeg(elt2_r);
        acc_i = FPAdd(acc_i, elt2_r, FPCR[]);
    Elem[result, 2 * p + 0, esize] = acc_r;
    Elem[result, 2 * p + 1, esize] = acc_i;

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



FCLAMP

Floating-point clamp to minimum/maximum number

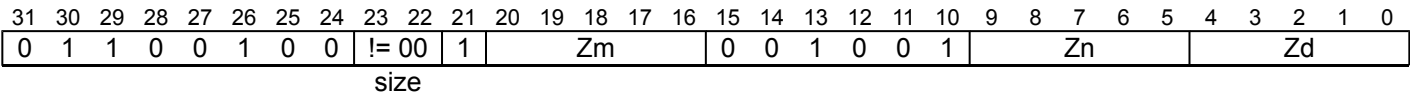
Clamp each floating-point element in the destination vector to between the floating-point minimum value in the corresponding element of the first source vector and the floating-point maximum value in the corresponding element of the second source vector and destructively place the clamped results in the corresponding elements of the destination vector.

Regardless of the value of FPCR.AH, the behavior is as follows for each mininum number and maximum number operation:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction is unpredicated.

SVE2
(FEAT\_SVE2p1)



FCLAMP <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) result;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[d, VL];

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    bits(esize) element3 = Elem[operand3, e, esize];
    Elem[result, e, esize] = FPMinNum(FPMaxNum(element1, element3, FPCR[]), element2, FPCR[]);
Z[d, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCM<cc> (vectors)

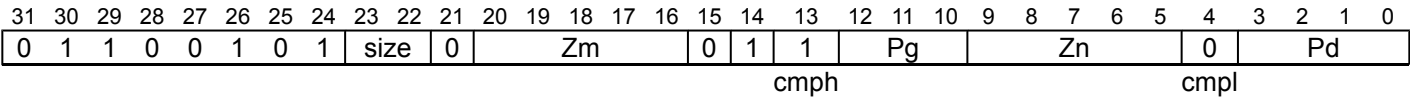
Floating-point compare vectors

Compare active floating-point elements in the first source vector with corresponding elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

<cc>	Comparison
EQ	equal
GE	greater than or equal
GT	greater than
NE	not equal
UO	unordered

This instruction is used by the pseudo-instructions [FCMLE \(vectors\)](#), and [FCMLT \(vectors\)](#).  
It has encodings from 5 classes: [Equal](#) , [Greater than](#) , [Greater than or equal](#) , [Not equal](#) and [Unordered](#)

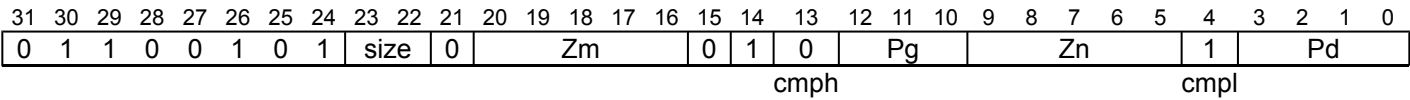
Equal



FCMEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_EQ;
```

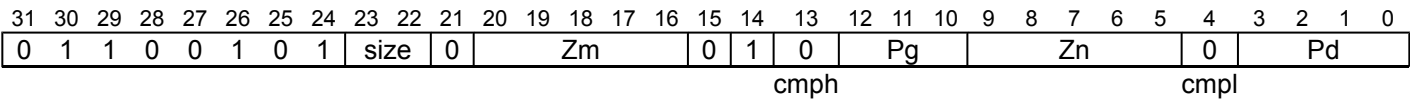
Greater than



FCMGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_GT;
```

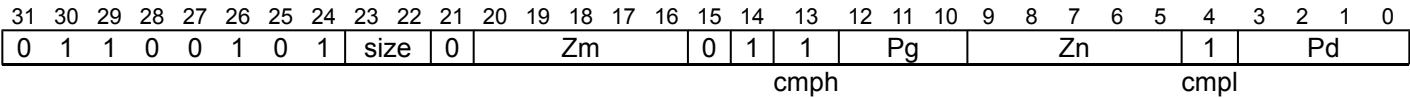
Greater than or equal



FCMGE <Pd>.<T>, <Pg>/z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_GE;
```

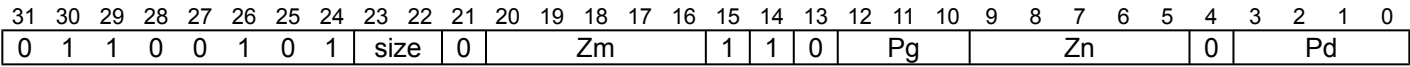
Not equal



FCMNE <Pd>.<T>, <Pg>/z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_NE;
```

Unordered



FCMUO <Pd>.<T>, <Pg>/z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_UN;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        boolean res;
        case op of
            when Cmp_EQ res = FPCmpareEQ(element1, element2, FPCR[]);
            when Cmp_GE res = FPCmpareGE(element1, element2, FPCR[]);
            when Cmp_GT res = FPCmpareGT(element1, element2, FPCR[]);
            when Cmp_UN res = FPCmpareUN(element1, element2, FPCR[]);
            when Cmp_NE res = FPCmpareNE(element1, element2, FPCR[]);
            when Cmp_LT res = FPCmpareGT(element2, element1, FPCR[]);
            when Cmp_LE res = FPCmpareGE(element2, element1, FPCR[]);
        bit pbit = if res then '1' else '0';
        Elem[result, e, psize] = ZeroExtend(pbit, psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

P[d, PL] = result;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCM<cc> (zero)

Floating-point compare vector with zero

Compare active floating-point elements in the source vector with zero, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

<cc>	Comparison
EQ	equal
GE	greater than or equal
GT	greater than
LE	less than or equal
LT	less than
NE	not equal
UO	unordered

It has encodings from 6 classes: [Equal](#) , [Greater than](#) , [Greater than or equal](#) , [Less than](#) , [Less than or equal](#) and [Not equal](#)

Equal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	1	0	0	1	0	0	0	1	Pg						Zn		0			Pd	
eq lt																ne															

FCMEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEComp op = Cmp_EQ;
```

Greater than

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	1	0	0	0	0	0	0	1	Pg						Zn		1			Pd	
eq lt																ne															

FCMGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEComp op = Cmp_GT;
```

Greater than or equal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	1	0	0	0	0	0	0	1	Pg						Zn		0			Pd	
eq lt																ne															

**FCMGE** <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEComp op = Cmp_GE;
```

## Less than

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	1	0	0	0	0	1	0	0	1	Pg									0			Pd
eq lt																ne															

**FCMLT** <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEComp op = Cmp_LT;
```

## Less than or equal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	1	0	0	0	0	1	0	0	1	Pg									1			Pd
eq lt																ne															

**FCMLE** <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEComp op = Cmp_LE;
```

## Not equal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	1	0	0	1	1	1	0	0	1	Pg									0			Pd
eq lt																ne															

**FCMNE** <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEComp op = Cmp_NE;
```

## Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        boolean res;
        case op of
            when Cmp_EQ res = FPCmpareEQ(element, 0<esize-1:0>, FPCR[]);
            when Cmp_GE res = FPCmpareGE(element, 0<esize-1:0>, FPCR[]);
            when Cmp_GT res = FPCmpareGT(element, 0<esize-1:0>, FPCR[]);
            when Cmp_NE res = FPCmpareNE(element, 0<esize-1:0>, FPCR[]);
            when Cmp_LT res = FPCmpareGT(0<esize-1:0>, element, FPCR[]);
            when Cmp_LE res = FPCmpareGE(0<esize-1:0>, element, FPCR[]);
        bit pbit = if res then '1' else '0';
        Elem[result, e, psize] = ZeroExtend(pbit, psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

P[d, PL] = result;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## FCMLA (indexed)

Floating-point complex multiply-add by indexed values with rotate

Multiply the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the floating-point complex numbers in each 128-bit segment of the first source vector by the specified complex number in the corresponding the second source vector segment rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

Then destructively add the products to the corresponding components of the complex numbers in the addend and destination vector, without intermediate rounding.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

The complex numbers within the second source vector are specified using an immediate index which selects the same complex number position within each 128-bit vector segment. The index range is from 0 to one less than the number of complex numbers per 128-bit segment, encoded in 1 to 2 bits depending on the size of the complex number. This instruction is unpredicated.

It has encodings from 2 classes: [Half-precision](#) and [Single-precision](#)

### Half-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	0	1	i2		Zm			0	0	0	1	rot				Zn					Zda		
size<1>size<0>																															

**FCMLA** <Zda>.H, <Zn>.H, <Zm>.H[<imm>], <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean neg_i = (rot<1> == '1');
boolean neg_r = (rot<0> != rot<1>);
```

### Single-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	1	1	i1	Zm			0			0	0	1	rot		Zn			Zda					
size<1>size<0>																															

**FCMLA** <Zda>.S, <Zn>.S, <Zm>.S[<imm>], <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean neg_i = (rot<1> == '1');
boolean neg_r = (rot<0> != rot<1>);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the half-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the single-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the half-precision variant: is the index of a Real and Imaginary pair, in the range 0 to 3, encoded in the "i2" field.  
For the single-precision variant: is the index of a Real and Imaginary pair, in the range 0 to 1, encoded in the "i1" field.
- <const> Is the const specifier, encoded in “rot”:

rot	<const>
00	#0
01	#90
10	#180
11	#270

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
constant integer pairspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for p = 0 to pairs-1
    segmentbase = p - (p MOD pairspersegment);
    s = segmentbase + index;
    addend_r = Elem[operand3, 2 * p + 0, esize];
    addend_i = Elem[operand3, 2 * p + 1, esize];
    elt1_a = Elem[operand1, 2 * p + sel_a, esize];
    elt2_a = Elem[operand2, 2 * s + sel_a, esize];
    elt2_b = Elem[operand2, 2 * s + sel_b, esize];
    if neg_r then elt2_a = FPNeg(elt2_a);
    if neg_i then elt2_b = FPNeg(elt2_b);
    addend_r = FPMulAdd(addend_r, elt1_a, elt2_a, FPCR[]);
    addend_i = FPMulAdd(addend_i, elt1_a, elt2_b, FPCR[]);
    Elem[result, 2 * p + 0, esize] = addend_r;
    Elem[result, 2 * p + 1, esize] = addend_i;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

FCMLA (vectors)

Floating-point complex multiply-add with rotate (predicated)

Multiply the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the floating-point complex numbers in the first source vector by the corresponding complex number in the second source vector rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation. Then destructively add the products to the corresponding components of the complex numbers in the addend and destination vector, without intermediate rounding. These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart. Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	size	0	Zm			0	rot	Pg		Zn			Zda											

FCMLA <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean neg_i = (rot<1> == '1');
boolean neg_r = (rot<0> != rot<1>);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in "rot":

rot	<const>
00	#0
01	#90
10	#180
11	#270

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for p = 0 to pairs-1
    addend_r = Elem[operand3, 2 * p + 0, esize];
    addend_i = Elem[operand3, 2 * p + 1, esize];
    if ActivePredicateElement(mask, 2 * p + 0, esize) then
        bits(esize) elt1_a = Elem[operand1, 2 * p + sel_a, esize];
        bits(esize) elt2_a = Elem[operand2, 2 * p + sel_a, esize];
        if neg_r then elt2_a = FPNeg(elt2_a);
        addend_r = FPMulAdd(addend_r, elt1_a, elt2_a, FPCR[]);
    if ActivePredicateElement(mask, 2 * p + 1, esize) then
        bits(esize) elt1_a = Elem[operand1, 2 * p + sel_a, esize];
        bits(esize) elt2_b = Elem[operand2, 2 * p + sel_b, esize];
        if neg_i then elt2_b = FPNeg(elt2_b);
        addend_i = FPMulAdd(addend_i, elt1_a, elt2_b, FPCR[]);
    Elem[result, 2 * p + 0, esize] = addend_r;
    Elem[result, 2 * p + 1, esize] = addend_i;

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCMLE (vectors)

Floating-point compare less than or equal to vector

Compare active floating-point elements in the first source vector being less than or equal to corresponding elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of [FCM<cc> \(vectors\)](#). This means:

- The encodings in this description are named to match the encodings of [FCM<cc> \(vectors\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [FCM<cc> \(vectors\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	Zm				0	1	0	Pg			Zn				0	Pd					
																cmph				cmpl											

FCMLE <Pd>.<T>, <Pg>/z, <Zm>.<T>, <Zn>.<T>

is equivalent to

FCMGE <Pd>.<T>, <Pg>/z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

- <Pd>

Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <T>

Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

- <Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of [FCM<cc> \(vectors\)](#) gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

FCMLT (vectors)

Floating-point compare less than vector

Compare active floating-point elements in the first source vector being less than corresponding elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of [FCM<cc> \(vectors\)](#). This means:

- The encodings in this description are named to match the encodings of [FCM<cc> \(vectors\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [FCM<cc> \(vectors\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	Zm				0	1	0	Pg			Zn				1	Pd					
																cmph				cmpl											

FCMLT <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

FCMGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of [FCM<cc> \(vectors\)](#) gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

FCPY

Copy 8-bit floating-point immediate to vector elements (predicated)

Copy a floating-point immediate into each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is used by the alias [FMOV \(immediate, predicated\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size		0	1	Pg				1	1	0	imm8								Zd				

FCPY <Zd>.<T>, <Pg>/M, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Zd);
bits(eseize) imm = VFPEExpandImm(imm8, esize);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<const> Is a floating-point immediate value expressible as  $\pm n \div 16 \times 2^r$ , where n and r are integers such that  $16 \leq n \leq 31$  and  $-3 \leq r \leq 4$ , i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the "imm8" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = imm;

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.





## FCVT

Floating-point convert precision (predicated)

Convert the size and precision of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

Since the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.

It has encodings from 6 classes: [Half-precision to single-precision](#) , [Half-precision to double-precision](#) , [Single-precision to half-precision](#) , [Single-precision to double-precision](#) , [Double-precision to half-precision](#) and [Double-precision to single-precision](#)

### Half-precision to single-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	1	0	0	0	1	0	0	1	1	0	1	Pg			Zn			Zd						

**FCVT** <Zd>.S, <Pg>/M, <Zn>.H

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 16;
constant integer d_esign = 32;
```

### Half-precision to double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	1	1	0	0	1	0	0	1	1	0	1	Pg			Zn			Zd						

**FCVT** <Zd>.D, <Pg>/M, <Zn>.H

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 16;
constant integer d_esign = 64;
```

### Single-precision to half-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	1	0	0	0	1	0	0	0	1	0	1	Pg			Zn			Zd						

**FCVT** <Zd>.H, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 32;
constant integer d_esign = 16;
```

Single-precision to double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	1	1	0	0	1	0	1	1	1	0	1	Pg			Zn			Zd						

FCVT <Zd>.D, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 32;
constant integer d_esign = 64;
```

Double-precision to half-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	1	1	0	0	1	0	0	0	1	0	1	Pg			Zn			Zd						

FCVT <Zd>.H, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 64;
constant integer d_esign = 16;
```

Double-precision to single-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	1	1	0	0	1	0	1	0	1	0	1	Pg			Zn			Zd						

FCVT <Zd>.S, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 64;
constant integer d_esign = 32;
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        bits(d_esize) res = FPConvertSVE(element<s_esize-1:0>, FPCR[], d_esize);
        Elem[result, e, esize] = ZeroExtend(res, esize);

Z[d, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTLT

Floating-point up convert long (top, predicated)

Convert odd-numbered floating-point elements from the source vector to the next higher precision, and place the results in the active overlapping double-width elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

It has encodings from 2 classes: [Half-precision to single-precision](#) and [Single-precision to double-precision](#)

Half-precision to single-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	0	0	0	1	0	0	1	1	0	1	Pg			Zn			Zd						

FCVTLT <Zd>.S, <Pg>/M, <Zn>.H

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Single-precision to double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	1	0	0	1	0	1	1	1	0	1	Pg			Zn			Zd						

FCVTLT <Zd>.D, <Pg>/M, <Zn>.S

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    bits(esize DIV 2) element = Elem[operand, 2*e + 1, esize DIV 2];
    Elem[result, e, esize] = FPConvertSVE(element, FPCR[], esize);

Z[d, VL] = result;
```



# FCVTNT

Floating-point down convert and narrow (top, predicated)

Convert active floating-point elements from the source vector to the next lower precision, and place the results in the odd-numbered half-width elements of the destination vector, leaving the even-numbered elements unchanged. Inactive elements in the destination vector register remain unmodified.

It has encodings from 2 classes: [Single-precision to half-precision](#) and [Double-precision to single-precision](#)

## Single-precision to half-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0	1	Pg			Zn			Zd						

FCVTNT <Zd>.H, <Pg>/M, <Zn>.S

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

## Double-precision to single-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	1	0	0	1	0	1	0	1	0	1	Pg			Zn			Zd						

FCVTNT <Zd>.S, <Pg>/M, <Zn>.D

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

## Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, 2*e + 1, esize DIV 2] = FPConvertSVE(element, FPCR[], esize DIV 2);

Z[d, VL] = result;
```



FCVTX

Floating-point down convert, rounding to odd (predicated)

Convert active double-precision floating-point elements from the source vector to single-precision, rounding to Odd, and place the results in the even-numbered 32-bit elements of the destination vector, while setting the odd-numbered elements to zero. Inactive elements in the destination vector register remain unmodified.

Rounding to Odd (aka Von Neumann rounding) permits a two-step conversion from double-precision to half-precision without incurring intermediate rounding errors.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	0	0	0	1	0	1	0	1	0	1	Pg			Zn			Zd							

FCVTX <Zd>.S, <Pg>/M, <Zn>.D

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 64;
constant integer d_esign = 32;
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        bits(d_esign) res = FPConvertSVE(element<s_esign-1:0>, FPCR[], FPRounding_ODD, d_esign);
        Elem[result, e, esize] = ZeroExtend(res, esize);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



# FCVTXNT

Floating-point down convert, rounding to odd (top, predicated)

Convert active double-precision floating-point elements from the source vector to single-precision, rounding to Odd, and place the results in the odd-numbered 32-bit elements of the destination vector, leaving the even-numbered elements unchanged. Inactive elements in the destination vector register remain unmodified.  
Rounding to Odd (aka Von Neumann rounding) permits a two-step conversion from double-precision to half-precision without incurring intermediate rounding errors.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	1	0	1		Pg												

FCVTXNT <Zd>.S, <Pg>/M, <Zn>.D

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

## Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, 2*e + 1, esize DIV 2] = FPConvertSVE(element, FPCR[], FPRounding_ODD, esize DIV 2);

Z[d, VL] = result;
```

FCVTZS

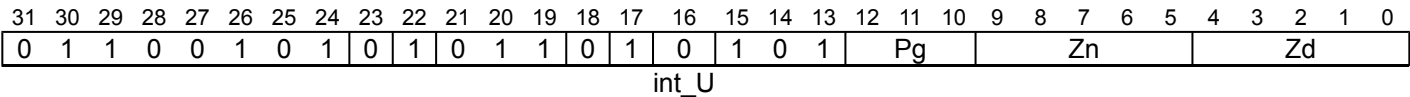
Floating-point convert to signed integer, rounding toward zero (predicated)

Convert to the signed integer nearer to zero from each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are sign-extended to fill each destination element.

It has encodings from 7 classes: [Half-precision to 16-bit](#) , [Half-precision to 32-bit](#) , [Half-precision to 64-bit](#) , [Single-precision to 32-bit](#) , [Single-precision to 64-bit](#) , [Double-precision to 32-bit](#) and [Double-precision to 64-bit](#)

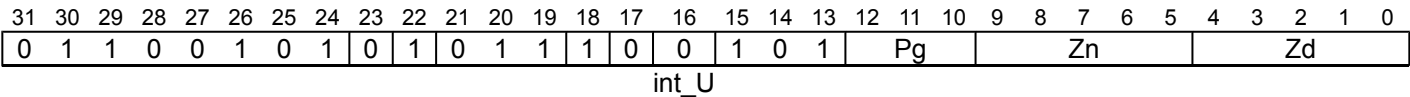
Half-precision to 16-bit



FCVTZS <Zd>.H, <Pg>/M, <Zn>.H

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 16;
constant integer d_esign = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

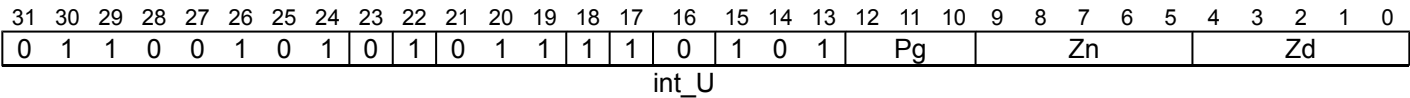
Half-precision to 32-bit



FCVTZS <Zd>.S, <Pg>/M, <Zn>.H

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 16;
constant integer d_esign = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

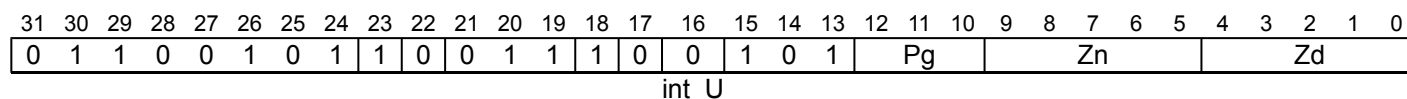
Half-precision to 64-bit



**FCVTZS <Zd>.D, <Pg>/M, <Zn>.H**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 16;
constant integer d_esign = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

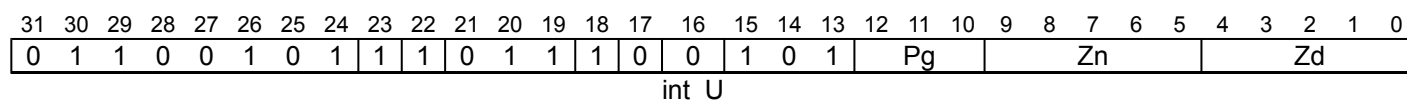
### Single-precision to 32-bit



**FCVTZS <Zd>.S, <Pg>/M, <Zn>.S**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 32;
constant integer d_esign = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

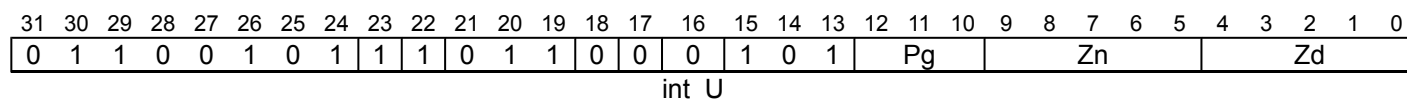
### Single-precision to 64-bit



**FCVTZS <Zd>.D, <Pg>/M, <Zn>.S**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 32;
constant integer d_esign = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

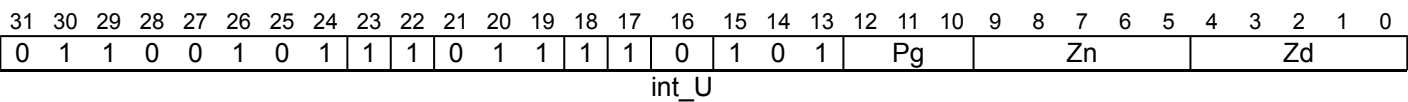
### Double-precision to 32-bit



**FCVTZS <Zd>.S, <Pg>/M, <Zn>.D**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 64;
constant integer d_esign = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

### Double-precision to 64-bit



**FCVTZS <Zd>.D, <Pg>/M, <Zn>.D**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 64;
constant integer d_esign = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

### Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    bits(esize) element = Elem[operand, e, esize];
    bits(d_esign) res = FPToFixed(element<s_esign-1:0>, 0, unsigned, FPCR[], rounding, d_esign);
    Elem[result, e, esize] = Extend(res, esize, unsigned);

Z[d, VL] = result;
```

### Operational information

- This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
  - The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FCVTZU

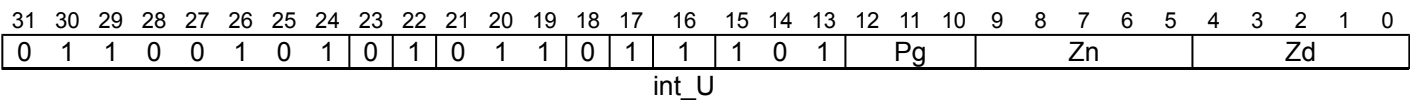
Floating-point convert to unsigned integer, rounding toward zero (predicated)

Convert to the unsigned integer nearer to zero from each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.

It has encodings from 7 classes: [Half-precision to 16-bit](#) , [Half-precision to 32-bit](#) , [Half-precision to 64-bit](#) , [Single-precision to 32-bit](#) , [Single-precision to 64-bit](#) , [Double-precision to 32-bit](#) and [Double-precision to 64-bit](#)

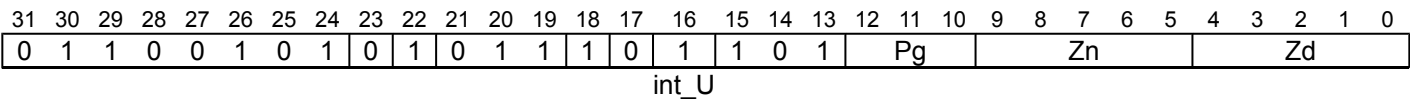
Half-precision to 16-bit



FCVTZU <Zd>.H, <Pg>/M, <Zn>.H

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 16;
constant integer d_esign = 16;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

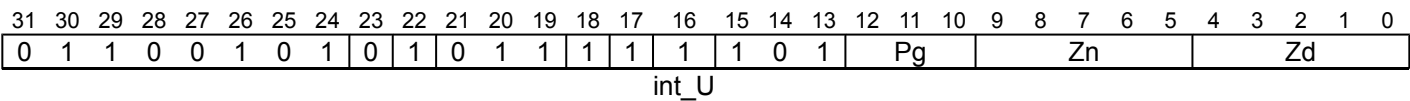
Half-precision to 32-bit



FCVTZU <Zd>.S, <Pg>/M, <Zn>.H

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 16;
constant integer d_esign = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

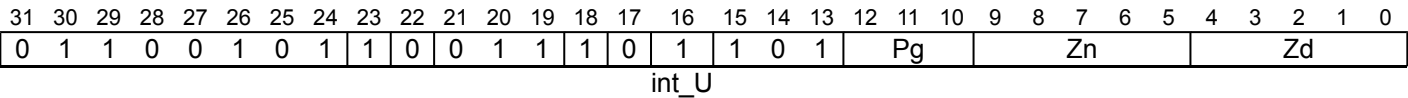
Half-precision to 64-bit



**FCVTZU <Zd>.D, <Pg>/M, <Zn>.H**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 16;
constant integer d_esign = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

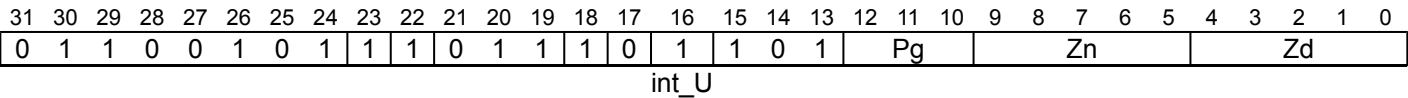
**Single-precision to 32-bit**



**FCVTZU <Zd>.S, <Pg>/M, <Zn>.S**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 32;
constant integer d_esign = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

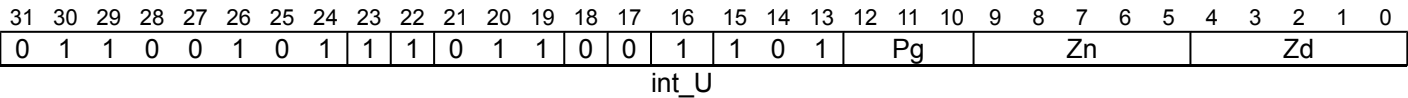
**Single-precision to 64-bit**



**FCVTZU <Zd>.D, <Pg>/M, <Zn>.S**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 32;
constant integer d_esign = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

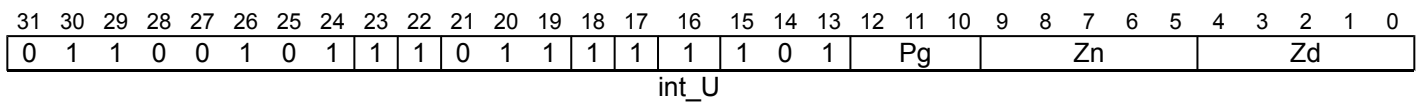
**Double-precision to 32-bit**



FCVTZU <Zd>.S, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 64;
constant integer d_esign = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

## Double-precision to 64-bit



FCVTZU <Zd>.D, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 64;
constant integer d_esign = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

## Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        bits(d_esign) res = FPToFixed(element<s_esign-1:0>, 0, unsigned, FPCR[], rounding, d_esign);
        Elem[result, e, esize] = Extend(res, esize, unsigned);

Z[d, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.



- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FDIV

Floating-point divide by vector (predicated)

Divide active floating-point elements of the first source vector by corresponding floating-point elements of the second source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	1	1	0	1	1	0	0	Pg			Zm					Zdn					

FDIV <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ActivePredicateElement(mask, e, esize) then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPDIV(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FDIVR

Floating-point reversed divide by vector (predicated)

Reversed divide active floating-point elements of the second source vector by corresponding floating-point elements of the first source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	1	1	0	0	1	0	0	Pg													

FDIVR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPDIV(element2, element1, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FDOT (indexed)

Half-precision floating-point indexed dot product

This instruction computes the fused sum-of-products of a pair of half-precision floating-point values held in each 32-bit element of the first source vector and a pair of half-precision floating-point values in an indexed 32-bit element of the second source vector, without intermediate rounding, and then destructively adds the single-precision sum-of-products to the corresponding single-precision element of the destination vector.

The half-precision floating-point pairs within the second source vector are specified using an immediate index which selects the same pair position within each 128-bit vector segment. The index range is from 0 to 3.

This instruction is unpredicated.

### SVE2

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	0	0	1	i2		Zm			0	1	0	0	0	0			Zn				Zda			

**FDOT** <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i2);
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index of a group of two 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
constant integer eltspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2 * s + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * s + 1, 16];
    bits(32) sum = Elem[operand3, e, 32];

    sum = FPDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FDOT (vectors)

Half-precision floating-point dot product

This instruction computes the fused sum-of-products of a pair of half-precision floating-point values held in each 32-bit element of the first source and second source vectors, without intermediate rounding, and then destructively adds the single-precision sum-of-products to the corresponding single-precision element of the destination vector. This instruction is unpredicated.

### SVE2 (FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	1	0	0	1	0	0	0	0	1	Zm					1	0	0	0	0	0	0	Zn					Zda				

FDOT <Zda>.S, <Zn>.H, <Zm>.H

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
    bits(32) sum = Elem[operand3, e, 32];

    sum = FPDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;

Z[da, VL] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



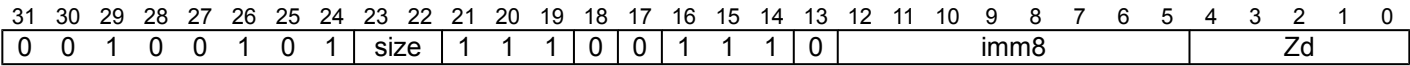


FDUP

Broadcast 8-bit floating-point immediate to vector elements (unpredicated)

Unconditionally broadcast the floating-point immediate into each element of the destination vector. This instruction is unpredicated.

This instruction is used by the alias [FMOV \(immediate, unpredicated\)](#).



FDUP <Zd>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer d = UInt(Zd);
bits(esize) imm = VFPEExpandImm(imm8, esize);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<const> Is a floating-point immediate value expressible as  $\pm n \div 16 \times 2^r$ , where n and r are integers such that  $16 \leq n \leq 31$  and  $-3 \leq r \leq 4$ , i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the "imm8" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) result;

for e = 0 to elements-1
    Elem[result, e, esize] = imm;

Z[d, VL] = result;
```

FEXPA

Floating-point exponential accelerator

The FEXPA instruction accelerates the polynomial series calculation of the EXP(x) function.

The double-precision variant copies the low 52 bits of an entry from a hard-wired table of 64-bit coefficients, indexed by the low 6 bits of each element of the source vector, and prepends to that the next 11 bits of the source element (src<16:6>), setting the sign bit to zero.

The single-precision variant copies the low 23 bits of an entry from hard-wired table of 32-bit coefficients, indexed by the low 6 bits of each element of the source vector, and prepends to that the next 8 bits of the source element (src<13:6>), setting the sign bit to zero.

The half-precision variant copies the low 10 bits of an entry from hard-wired table of 16-bit coefficients, indexed by the low 5 bits of each element of the source vector, and prepends to that the next 5 bits of the source element (src<9:5>), setting the sign bit to zero.

A coefficient table entry with index m holds the floating-point value 2<sup>(m/64)</sup>, or for the half-precision variant 2<sup>(m/32)</sup>. This instruction is unpredicated.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1	0	0	0	0	0	0	1	0	1	1	1	0	Zn				Zd					

FEXPA <Zd>.<T>, <Zn>.<T>

```
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPExpA(element);

Z[d, VL] = result;
```

FLOGB

Floating-point base 2 logarithm as integer

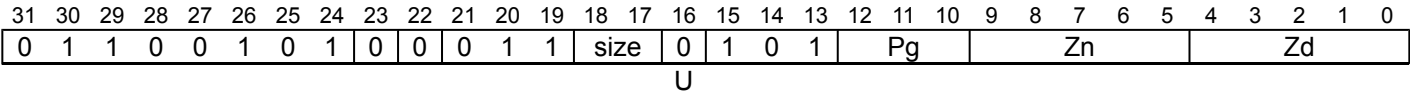
This instruction returns the signed integer base 2 logarithm of each floating-point input element |x| after normalization.

This is the unbiased exponent of x used in the representation of the floating-point value, such that, for positive x,  $x = \text{significand} \times 2^{\text{exponent}}$ .

The integer results are placed in elements of the destination vector which have the same width (ESIZE) as the floating-point input elements:

- If x is normal, the result is the base 2 logarithm of x.
- If x is subnormal, the result corresponds to the normalized representation.
- If x is infinite, the result is  $2^{(\text{esize}-1)}-1$ .
- If x is  $\pm 0.0$  or NaN, the result is  $-2^{(\text{esize}-1)}$ .

Inactive elements in the destination vector register remain unmodified.



FLOGB <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = FPLogB(element, FPCR[]);

Z[d, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

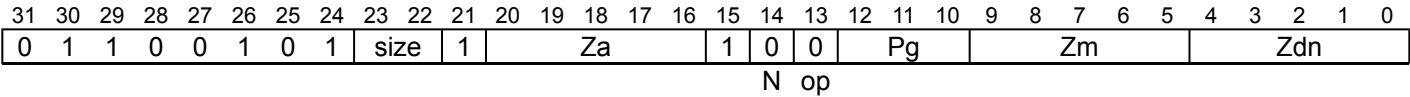
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAD

Floating-point fused multiply-add vectors (predicated), writing multiplicand [ $Z_{dn} = Z_a + Z_{dn} * Z_m$ ]

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third (addend) vector without intermediate rounding. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.



```
FMAD <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Za> Is the name of the third source scalable vector register, encoded in the "Za" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];

        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAX (immediate)

Floating-point maximum with immediate (predicated)

Determine the maximum of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if the element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if the element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both the element and the immediate are zeros, regardless of the sign of either zero, the result is the immediate.
- If the element is a NaN, regardless of the value of FPCR.DN, the result is the immediate.

Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	1	1	1	1	0	1	0	0	Pg	0	0	0	0	i1	Zdn							

FMAX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros(esize) else FPOne('0', esize);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in "i1":

i1	<const>
0	#0.0
1	#1.0



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = FMax(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAX (vectors)

Floating-point maximum (predicated)

Determine the maximum of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	!= 00	0	0	0	1	1	0	1	0	0	Pg				Zm					Zdn				
size																															

FMAX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ActivePredicateElement(mask, e, esize) then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMMax(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAXNM (immediate)

Floating-point maximum number with immediate (predicated)

Determine the maximum number value of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If the element is a quiet NaN, the result is the immediate value.
- When FPCR.DN is 0, if the element is a signaling NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if the element is a signaling NaN, the result is Default NaN.

Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	1	1	1	0	0	1	0	0	Pg	0	0	0	0	i1								Zdn

FMAXNM <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros(esize) else FPOne('0', esize);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in "i1":

i1	<const>
0	#0.0
1	#1.0

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = FPMaxNum(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAXNM (vectors)

Floating-point maximum number (predicated)

Determine the maximum number value of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	!= 00	0	0	0	1	0	0	1	0	0		Pg												
size																Zm				Zdn											

FMAXNM <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ActivePredicateElement(mask, e, esize) then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMaXNum(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAXNMP

Floating-point maximum number pairwise

Compute the maximum value of each pair of adjacent floating-point elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector. Regardless of the value of FPCR.AH, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	size	0	1	0	1	0	0	0	1	0	0	Pg	Zm			Zdn								

FMAXNMP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result = Z[dn, VL];
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        if IsEven(e) then
            element1 = Elem[operand1, e + 0, esize];
            element2 = Elem[operand1, e + 1, esize];
        else
            element1 = Elem[operand2, e - 1, esize];
            element2 = Elem[operand2, e + 0, esize];
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAXNMQV

Floating-point maximum number recursive reduction of quadword vector segments

Floating-point maximum number of the same element numbers from each 128-bit source vector segment using a recursive pairwise reduction, placing each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as the default NaN.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

SVE2
(FEAT\_SVE2p1)

Table with 32 columns (bit positions 31 to 0) and 3 rows. Row 1: Bit positions. Row 2: Bit values (0 or 1). Row 3: Field names (size, Pg, Zn, Vd).

FMAXNMQV <Vd>.<T>, <Pg>, <Zn>.<Tb>

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<Vd> Is the name of the destination SIMD&FP register, encoded in the "Vd" field.

<T> Is an arrangement specifier, encoded in "size":

Table with 2 columns: size, <T>. Rows: 00 RESERVED, 01 8H, 10 4S, 11 2D.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

Table with 2 columns: size, <Tb>. Rows: 00 RESERVED, 01 H, 10 S, 11 D.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elemperssegment = 128 DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPDefaultNaN(esize);
bits(128) result = Zeros(128);

constant integer p2bits = CeilPow2(segments*esize);
constant integer p2elems = p2bits DIV esize;

for e = 0 to elemperssegment-1
    bits(p2bits) stmp;
    bits(esize) dtmp;
    for s = 0 to p2elems-1
        if s < segments && ActivePredicateElement(mask, s * elemperssegment + e, esize) then
            Elem[stmp, s, esize] = Elem[operand, s * elemperssegment + e, esize];
        else
            Elem[stmp, s, esize] = identity;
    dtmp = Reduce(ReduceOp\_FMAXNUM, stmp, esize);
    Elem[result, e, esize] = dtmp;
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAXNMV

Floating-point maximum number recursive reduction to scalar

Floating-point maximum number horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the default NaN.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	0	1	0	0	0	0	1	Pg			Zn			Vd							

FMAXNMV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPDefaultNaN(esize);

V[d, esize] = ReducePredicated(ReduceOp_FMAXNUM, operand, mask, identity);
```

FMAXP

Floating-point maximum pairwise

Compute the maximum value of each pair of adjacent floating-point elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

When FPCR.AH is 0, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows for each pairwise operation:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	size		0	1	0	1	1	0	1	0	0	Pg			Zm				Zdn					

FMAXP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result = Z[dn, VL];
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        if IsEven(e) then
            element1 = Elem[operand1, e + 0, esize];
            element2 = Elem[operand1, e + 1, esize];
        else
            element1 = Elem[operand2, e - 1, esize];
            element2 = Elem[operand2, e + 0, esize];
        Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAXQV

Floating-point maximum reduction of quadword vector segments

Floating-point maximum of the same element numbers from each 128-bit source vector segment using a recursive pairwise reduction, placing each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as -Infinity.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either value is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of FPCR.DN, the result is the second value.

SVE2
(FEAT\_SVE2p1)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 0 size 0 1 0 1 1 0 1 0 1 Pg Zn Vd

FMAXQV <Vd>.<T>, <Pg>, <Zn>.<Tb>

if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler Symbols

<Vd> Is the name of the destination SIMD&FP register, encoded in the "Vd" field.

<T> Is an arrangement specifier, encoded in "size":

Table with 2 columns: size, <T>. Rows: 00 RESERVED, 01 8H, 10 4S, 11 2D.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

Table with 2 columns: size, <Tb>. Rows: 00 RESERVED, 01 H, 10 S, 11 D.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elemperssegment = 128 DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPInfinity('1', esize);
bits(128) result = Zeros(128);

constant integer p2bits = CeilPow2(segments*esize);
constant integer p2elems = p2bits DIV esize;

for e = 0 to elemperssegment-1
    bits(p2bits) stmp;
    bits(esize) dtmp;
    for s = 0 to p2elems-1
        if s < segments && ActivePredicateElement(mask, s * elemperssegment + e, esize) then
            Elem[stmp, s, esize] = Elem[operand, s * elemperssegment + e, esize];
        else
            Elem[stmp, s, esize] = identity;
    dtmp = Reduce(ReduceOp\_FMAX, stmp, esize);
    Elem[result, e, esize] = dtmp;
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



FMAXV

Floating-point maximum recursive reduction to scalar

Floating-point maximum horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as -Infinity.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either value is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of FPCR.DN, the result is the second value.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	0	1	1	0	0	0	1	Pg			Zn				Vd						

FMAXV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPInfinity('1', esize);

V[d, esize] = ReducePredicated(ReduceOp_FMAX, operand, mask, identity);
```



FMIN (immediate)

Floating-point minimum with immediate (predicated)

Determine the minimum of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if the element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if the element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both the element and the immediate are zeros, regardless of the sign of either zero, the result is the immediate.
- If the element is a NaN, regardless of the value of FPCR.DN, the result is the immediate.

Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	1	1	1	1	1	1	0	0	Pg	0	0	0	0	i1	Zdn							

FMIN <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros(esize) else FPOne('0', esize);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in "i1":

i1	<const>
0	#0.0
1	#1.0

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = FPMIn(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMIN (vectors)

Floating-point minimum (predicated)

Determine the minimum of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	!= 00	0	0	0	1	1	1	1	0	0		Pg					Zm					Zdn		
																size															

FMIN <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ActivePredicateElement(mask, e, esize) then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMIn(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMINNM (immediate)

Floating-point minimum number with immediate (predicated)

Determine the minimum number value of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If the element is a quiet NaN, the result is the immediate value.
- When FPCR.DN is 0, if the element is a signaling NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if the element is a signaling NaN, the result is Default NaN.

Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	1	1	1	0	1	1	0	0	Pg	0	0	0	0	i1	Zdn							

FMINNM <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esModule) imm = if i1 == '0' then Zeros(esModule) else FPOne('0', esize);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in "i1":

i1	<const>
0	#0.0
1	#1.0

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = FPMinNum(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



FMINNM (vectors)

Floating-point minimum number (predicated)

Determine the minimum number value of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	!= 00	0	0	0	1	0	1	1	0	0		Pg												
size																Zm				Zdn											

FMINNM <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ActivePredicateElement(mask, e, esize) then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMINNMP

Floating-point minimum number pairwise

Compute the minimum value of each pair of adjacent floating-point elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector. Regardless of the value of FPCR.AH, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	size		0	1	0	1	0	1	1	0	0	Pg			Zm				Zdn					

FMINNMP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result = Z[dn, VL];
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        if IsEven(e) then
            element1 = Elem[operand1, e + 0, esize];
            element2 = Elem[operand1, e + 1, esize];
        else
            element1 = Elem[operand2, e - 1, esize];
            element2 = Elem[operand2, e + 0, esize];
        Elem[result, e, esize] = FPCMinNum(element1, element2, FPCR[]);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMINNMQV

Floating-point minimum number recursive reduction of quadword vector segments

Floating-point minimum number of the same element numbers from each 128-bit source vector segment using a recursive pairwise reduction, placing each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as the default NaN.  
Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

SVE2  
(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	size	0	1	0	1	0	1	1	0	1	Pg				Zn				Vd					

FMINNMQV <Vd>.<T>, <Pg>, <Zn>.<Tb>

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<Vd> Is the name of the destination SIMD&FP register, encoded in the "Vd" field.

<T> Is an arrangement specifier, encoded in "size":

size	<T>
00	RESERVED
01	8H
10	4S
11	2D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	H
10	S
11	D

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elemperssegment = 128 DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPDefaultNaN(esize);
bits(128) result = Zeros(128);

constant integer p2bits = CeilPow2(segments*esize);
constant integer p2elems = p2bits DIV esize;

for e = 0 to elemperssegment-1
    bits(p2bits) stmp;
    bits(esize) dtmp;
    for s = 0 to p2elems-1
        if s < segments && ActivePredicateElement(mask, s * elemperssegment + e, esize) then
            Elem[stmp, s, esize] = Elem[operand, s * elemperssegment + e, esize];
        else
            Elem[stmp, s, esize] = identity;
    dtmp = Reduce(ReduceOp\_FMINNUM, stmp, esize);
    Elem[result, e, esize] = dtmp;
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMINNMV

Floating-point minimum number recursive reduction to scalar

Floating-point minimum number horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the default NaN.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	0	1	0	1	0	0	1	Pg	Zn			Vd									

FMINNMV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPDefaultNaN(esize);

V[d, esize] = ReducePredicated(ReduceOp_FMINNUM, operand, mask, identity);
```

FMINP

Floating-point minimum pairwise

Compute the minimum value of each pair of adjacent floating-point elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

When FPCR.AH is 0, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows for each pairwise operation:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	size			0	1	0	1	1	1	1	0	0	Pg			Zm			Zdn					

FMINP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result = Z[dn, VL];
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        if IsEven(e) then
            element1 = Elem[operand1, e + 0, esize];
            element2 = Elem[operand1, e + 1, esize];
        else
            element1 = Elem[operand2, e - 1, esize];
            element2 = Elem[operand2, e + 0, esize];
        Elem[result, e, esize] = FPMIn(element1, element2, FPCR[]);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMINQV

Floating-point minimum recursive reduction of quadword vector segments

Floating-point minimum of the same element numbers from each 128-bit source vector segment using a recursive pairwise reduction, placing each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as +Infinity.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either value is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of FPCR.DN, the result is the second value.

SVE2
(FEAT\_SVE2p1)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 0 size 0 1 0 1 1 1 1 0 1 Pg Zn Vd

FMINQV <Vd>.<T>, <Pg>, <Zn>.<Tb>

if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler Symbols

<Vd> Is the name of the destination SIMD&FP register, encoded in the "Vd" field.

<T> Is an arrangement specifier, encoded in "size":

Table with 2 columns: size, <T>. Rows: 00 RESERVED, 01 8H, 10 4S, 11 2D.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

Table with 2 columns: size, <Tb>. Rows: 00 RESERVED, 01 H, 10 S, 11 D.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elemperssegment = 128 DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPInfinity('0', esize);
bits(128) result = Zeros(128);

constant integer p2bits = CeilPow2(segments*esize);
constant integer p2elems = p2bits DIV esize;

for e = 0 to elemperssegment-1
    bits(p2bits) stmp;
    bits(esize) dtmp;
    for s = 0 to p2elems-1
        if s < segments && ActivePredicateElement(mask, s * elemperssegment + e, esize) then
            Elem[stmp, s, esize] = Elem[operand, s * elemperssegment + e, esize];
        else
            Elem[stmp, s, esize] = identity;
    dtmp = Reduce(ReduceOp\_FMIN, stmp, esize);
    Elem[result, e, esize] = dtmp;
V[d, 128] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMINV

Floating-point minimum recursive reduction to scalar

Floating-point minimum horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as +Infinity. When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either value is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of FPCR.DN, the result is the second value.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	0	1	1	1	0	0	1	Pg			Zn				Vd						

FMINV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPInfinity('0', esize);

V[d, esize] = ReducePredicated(ReduceOp_FMIN, operand, mask, identity);
```



## FMLA (indexed)

Floating-point fused multiply-add by indexed elements ( $Zda = Zda + Zn * Zm[indexed]$ )

Multiply all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively added without intermediate rounding to the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

It has encodings from 3 classes: [Half-precision](#) , [Single-precision](#) and [Double-precision](#)

### Half-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	0	i3h	1	i3l	Zm			0	0	0	0	0	0	Zn				Zda						
																					op										

**FMLA** <Zda>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;
```

### Single-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	0	1	i2	Zm			0	0	0	0	0	0	Zn				Zda						
size<1>size<0>											op																				

**FMLA** <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;
```

### Double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	1	1	i1	Zm			0	0	0	0	0	0	Zn				Zda						
size<1>size<0>												op																			

**FMLA** <Zda>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(il);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;
```

## Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the half-precision and single-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  For the double-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the half-precision variant: is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  For the single-precision variant: is the immediate index, in the range 0 to 3, encoded in the "i2" field.  For the double-precision variant: is the immediate index, in the range 0 to 1, encoded in the "i1" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, s, esize];
    bits(esize) element3 = Elem[result, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMLA (vectors)

Floating-point fused multiply-add vectors (predicated), writing addend [Zda = Zda + Zn \* Zm]

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third source (addend) vector without intermediate rounding. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	!= 00	1	Zm						0	0	0	Pg			Zn				Zda					
size										N op																					

FMLA <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    bits(esize) element3 = Elem[operand3, e, esize];

    if op1_neg then element1 = FPNeg(element1);
    if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = Elem[operand3, e, esize];

Z[da, VL] = result;
```



## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMLALB (indexed)

Half-precision floating-point multiply-add long to single-precision (bottom, indexed)

This half-precision floating-point multiply-add long instruction widens the even-numbered half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the first source vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	0	1	i3h	Zm	0	1	0	0	i3l	0	Zn				Zda								
o2											op							T													

**FMLALB** <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
boolean opl_neg = FALSE;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 0, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, s, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if opl_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

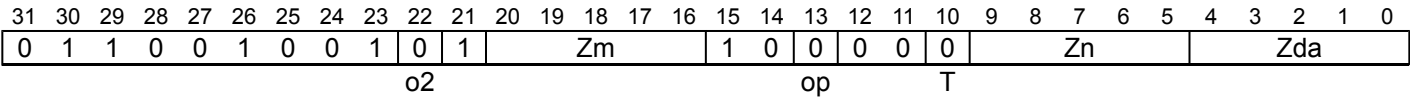
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMLALB (vectors)

Half-precision floating-point multiply-add long to single-precision (bottom)

This half-precision floating-point multiply-add long instruction widens the even-numbered half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the source vectors. This instruction is unpredicated.



**FMLALB** <Zda>.S, <Zn>.H, <Zm>.H

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean opl_neg = FALSE;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 0, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, 2 * e + 0, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if opl_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

## FMLALT (indexed)

Half-precision floating-point multiply-add long to single-precision (top, indexed)

This half-precision floating-point multiply-add long instruction widens the odd-numbered half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the first source vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	0	1	i3h	Zm	0	1	0	0	i3l	1	Zn				Zda								
o2											op							T													

**FMLALT** <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
boolean opl_neg = FALSE;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 1, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, s, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if opl_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

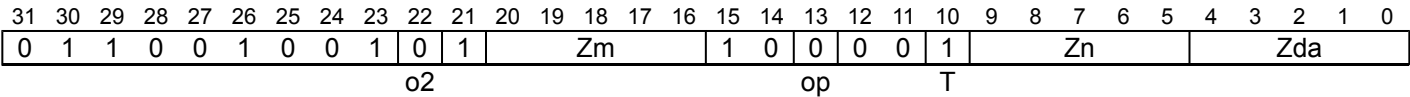
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# FMLALT (vectors)

Half-precision floating-point multiply-add long to single-precision (top)

This half-precision floating-point multiply-add long instruction widens the odd-numbered half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the source vectors. This instruction is unpredicated.



FMLALT <Zda>.S, <Zn>.H, <Zm>.H

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean opl_neg = FALSE;
```

## Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 1, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, 2 * e + 1, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if opl_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

## FMLS (indexed)

Floating-point fused multiply-subtract by indexed elements ( $Zda = Zda + -Zn * Zm[\text{indexed}]$ )

Multiply all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively subtracted without intermediate rounding from the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

It has encodings from 3 classes: [Half-precision](#) , [Single-precision](#) and [Double-precision](#)

### Half-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
0	1	1	0	0	1	0	0	0	i3h	1	i3l	Zm			0	0	0	0	0	1	Zn					Zda													
																					op																		

**FMLS** <Zda>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;
```

### Single-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	0	1	i2	Zm			0	0	0	0	0	1	Zn					Zda					
size<1>size<0>											op																				

**FMLS** <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;
```

### Double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	1	1	i1	Zm			0	0	0	0	0	1	Zn					Zda					
size<1>size<0>											op																				



**FMLS** <Zda>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(il);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;
```

## Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the half-precision and single-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  For the double-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the half-precision variant: is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  For the single-precision variant: is the immediate index, in the range 0 to 3, encoded in the "i2" field.  For the double-precision variant: is the immediate index, in the range 0 to 1, encoded in the "i1" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, s, esize];
    bits(esize) element3 = Elem[result, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

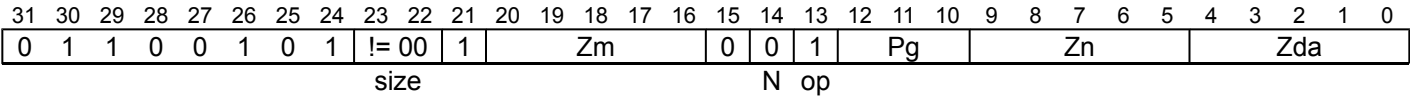
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMLS (vectors)

Floating-point fused multiply-subtract vectors (predicated), writing addend [Zda = Zda + -Zn \* Zm]

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third source (addend) vector without intermediate rounding. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.



```
FMLS <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    bits(esize) element3 = Elem[operand3, e, esize];

    if op1_neg then element1 = FPNeg(element1);
    if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = Elem[operand3, e, esize];

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

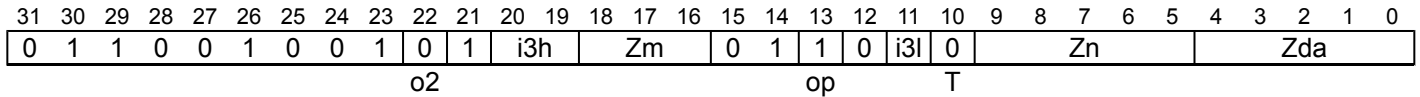
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMLSLB (indexed)

Half-precision floating-point multiply-subtract long from single-precision (bottom, indexed)

This half-precision floating-point multiply-subtract long instruction widens the even-numbered half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the first source vector. This instruction is unpredicated.



**FMLSLB** <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
boolean opl_neg = TRUE;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 0, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, s, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if opl_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

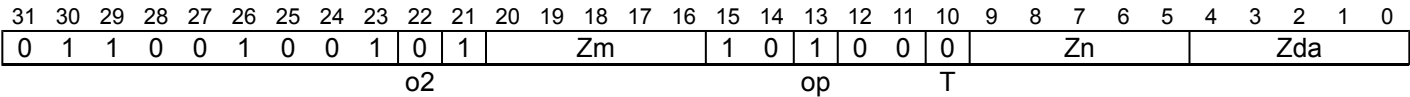
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# FMLSILB (vectors)

Half-precision floating-point multiply-subtract long from single-precision (bottom)

This half-precision floating-point multiply-subtract long instruction widens the even-numbered half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the source vectors. This instruction is unpredicated.



FMLSILB <Zda>.S, <Zn>.H, <Zm>.H

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean opl_neg = TRUE;
```

## Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 0, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, 2 * e + 0, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if opl_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

## Operational information

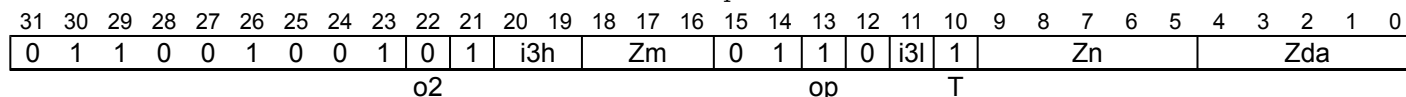
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

## FMLSLT (indexed)

Half-precision floating-point multiply-subtract long from single-precision (top, indexed)

This half-precision floating-point multiply-subtract long instruction widens the odd-numbered half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the first source vector. This instruction is unpredicated.



**FMLSLT** <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
boolean opl_neg = TRUE;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 1, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, s, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if opl_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

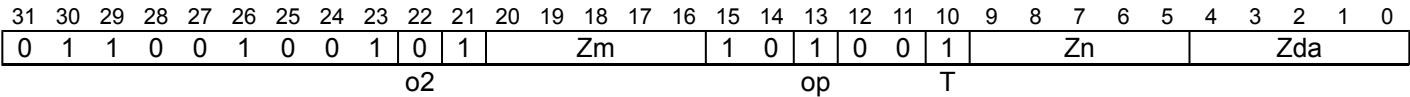
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# FMLSLT (vectors)

Half-precision floating-point multiply-subtract long from single-precision (top)

This half-precision floating-point multiply-subtract long instruction widens the odd-numbered half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the source vectors. This instruction is unpredicated.



```
FMLSLT <Zda>.S, <Zn>.H, <Zm>.H
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean opl_neg = TRUE;
```

## Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 1, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, 2 * e + 1, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if opl_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

# FMMLA

Floating-point matrix multiply-accumulate

The floating-point matrix multiply-accumulate instruction supports single-precision and double-precision data types in a 2×2 matrix contained in segments of 128 or 256 bits, respectively. It multiplies the 2×2 matrix in each segment of the first source vector by the 2×2 matrix in the corresponding segment of the second source vector. The resulting 2×2 matrix product is then destructively added to the matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing a 2-way dot product per destination element. This instruction is unpredicated. The single-precision variant is vector length agnostic. The double-precision variant requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.

ID\_AA64ZFR0\_EL1.F32MM indicates whether the single-precision variant is implemented.

ID\_AA64ZFR0\_EL1.F64MM indicates whether the double-precision variant is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

## 32-bit element

(FEAT\_F32MM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	0	1	Zm					1	1	1	0	0	1	Zn					Zda				

FMMLA <Zda>.S, <Zn>.S, <Zm>.S

```
if !HaveSVE() || !HaveSVEFP32MatMulExt() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

## 64-bit element

(FEAT\_F64MM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	1	1	Zm					1	1	1	0	0	1	Zn					Zda				

FMMLA <Zda>.D, <Zn>.D, <Zm>.D

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 64;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

## Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < esize * 4 then UNDEFINED;
constant integer segments = VL DIV (4 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result = Zeros(VL);
bits(4*esize) op1, op2;
bits(4*esize) res, addend;

for s = 0 to segments-1
    op1    = Elem[operand1, s, 4*esize];
    op2    = Elem[operand2, s, 4*esize];
    addend = Elem[operand3, s, 4*esize];
    res    = FPMatMulAdd(addend, op1, op2, esize, FPCR[]);
    Elem[result, s, 4*esize] = res;

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMOV (immediate, predicated)

Move 8-bit floating-point immediate to vector elements (predicated)

Move a floating-point immediate into each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of [FCPY](#). This means:

- The encodings in this description are named to match the encodings of [FCPY](#).
- The description of [FCPY](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size		0	1	Pg				1	1	0	imm8								Zd				

FMOV <Zd>.<T>, <Pg>/M, #<const>

is equivalent to

FCPY <Zd>.<T>, <Pg>/M, #<const>

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<const> Is a floating-point immediate value expressible as  $\pm n \div 16 \times 2^r$ , where n and r are integers such that  $16 \leq n \leq 31$  and  $-3 \leq r \leq 4$ , i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the "imm8" field.

Operation

The description of [FCPY](#) gives the operational pseudocode for this instruction.

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMOV (immediate, unpredicated)

Move 8-bit floating-point immediate to vector elements (unpredicated)

Unconditionally broadcast the floating-point immediate into each element of the destination vector. This instruction is unpredicated.

This is an alias of [FDUP](#). This means:

- The encodings in this description are named to match the encodings of [FDUP](#).
- The description of [FDUP](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	1	1	0	0	1	1	1	0	imm8								Zd					

FMOV <Zd>.<T>, #<const>

is equivalent to

[FDUP](#) <Zd>.<T>, #<const>

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<const> Is a floating-point immediate value expressible as  $\pm n \div 16 \times 2^r$ , where n and r are integers such that  $16 \leq n \leq 31$  and  $-3 \leq r \leq 4$ , i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the "imm8" field.

Operation

The description of [FDUP](#) gives the operational pseudocode for this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMOV (zero, predicated)

Move floating-point +0.0 to vector elements (predicated)

Move floating-point constant +0.0 to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is a pseudo-instruction of [CPY \(immediate, merging\)](#). This means:

- The encodings in this description are named to match the encodings of [CPY \(immediate, merging\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [CPY \(immediate, merging\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size		0	1	Pg				0	1	0	0	0	0	0	0	0	0	0	Zd				
M sh																imm8															

FMOV <Zd>.<T>, <Pg>/M, #0.0

is equivalent to

CPY <Zd>.<T>, <Pg>/M, #0

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Operation

The description of [CPY \(immediate, merging\)](#) gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



FMOV (zero, unpredicated)

Move floating-point +0.0 to vector elements (unpredicated)

Unconditionally broadcast the floating-point constant +0.0 into each element of the destination vector. This instruction is unpredicated.

This is a pseudo-instruction of [DUP \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [DUP \(immediate\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [DUP \(immediate\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	Zd				
sh																		imm8													

FMOV <Zd>.<T>, #0.0

is equivalent to

DUP <Zd>.<T>, #0

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

The description of [DUP \(immediate\)](#) gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

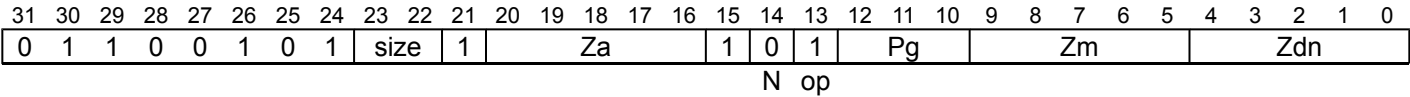
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



FMSB

Floating-point fused multiply-subtract vectors (predicated), writing multiplicand [Zdn = Za + -Zdn \* Zm]

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third (addend) vector without intermediate rounding. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.



FMSB <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Za> Is the name of the third source scalable vector register, encoded in the "Za" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];

        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMUL (immediate)

Floating-point multiply by immediate (predicated)

Multiply by an immediate each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +2.0 only. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	1	1	0	1	0	1	0	0	Pg	0	0	0	0	i1	Zdn							

FMUL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esome) imm = if i1 == '0' then FPPointFive('0', esize) else FPTwo('0', esize);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in "i1":

i1	<const>
0	#0.5
1	#2.0

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = FPMul(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# FMUL (indexed)

Floating-point multiply by indexed elements

Multiply all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The results are placed in the corresponding elements of the destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

It has encodings from 3 classes: [Half-precision](#) , [Single-precision](#) and [Double-precision](#)

## Half-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	0	i3h	1	i3l	Zm			0	0	1	0	0	0	Zn				Zd						

FMUL <Zd>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

## Single-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	0	1	i2	Zm			0	0	1	0	0	0	Zn				Zd						
size<1>size<0>																															

FMUL <Zd>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

## Double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	1	1	i1	Zm			0	0	1	0	0	0	Zn				Zd						
size<1>size<0>																															

FMUL <Zd>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

## Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

- <Zm> For the half-precision and single-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the double-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the half-precision variant: is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the single-precision variant: is the immediate index, in the range 0 to 3, encoded in the "i2" field.  
For the double-precision variant: is the immediate index, in the range 0 to 1, encoded in the "i1" field.

## Operation

```

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, s, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);

Z[d, VL] = result;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# FMUL (vectors, predicated)

Floating-point multiply vectors (predicated)

Multiply active floating-point elements of the first source vector by corresponding floating-point elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	!= 00	0	0	0	0	1	0	1	0	0		Pg						Zm					Zdn	
								size																							

FMUL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

## Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

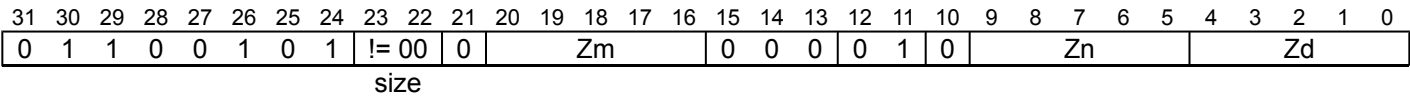
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



FMUL (vectors, unpredicated)

Floating-point multiply vectors (unpredicated)

Multiply all elements of the first source vector by corresponding floating-point elements of the second source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.



```
FMUL <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);

Z[d, VL] = result;
```

FMULX

Floating-point multiply-extended vectors (predicated)

Multiply active floating-point elements of the first source vector by corresponding floating-point elements of the second source vector except that  $\infty \times 0.0$  gives 2.0 instead of NaN, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

The instruction can be used with FRECPX to safely convert arbitrary elements in mathematical vector space to UNIT VECTORS or DIRECTION VECTORS with length 1.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	1	0	1	0	1	0	0	0	Pg					Zm					Zdn		

FMULX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ActivePredicateElement(mask, e, esize) then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMulX(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# FNEG

Floating-point negate (predicated)

Negate each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This inverts the sign bit and cannot signal a floating-point exception. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	1	0	1	1	0	1	Pg			Zn				Zd						

FNEG <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

## Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = FPNeg(element);

Z[d, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

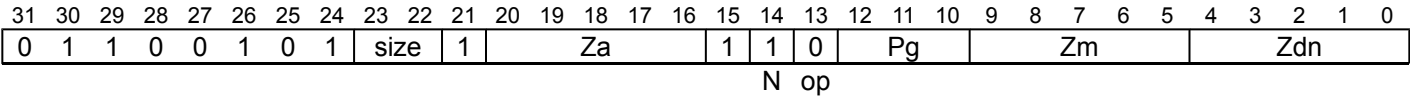
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



FNMAD

Floating-point negated fused multiply-add vectors (predicated), writing multiplicand [Zdn = -Za + -Zdn \* Zm]

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third (addend) vector without intermediate rounding. Destructively place the negated results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.



```
FNMAD <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean op1_neg = TRUE;
boolean op3_neg = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Za> Is the name of the third source scalable vector register, encoded in the "Za" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];

        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

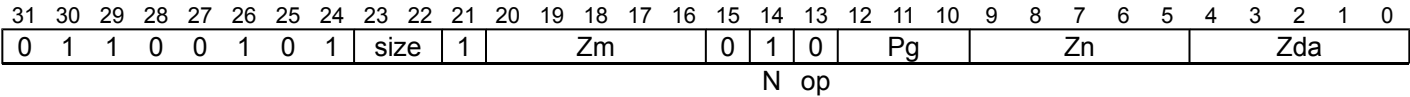
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FNMLA

Floating-point negated fused multiply-add vectors (predicated), writing addend [Zda = -Zda + -Zn \* Zm]

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third source (addend) vector without intermediate rounding. Destructively place the negated results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.



```
FNMLA <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = TRUE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];

        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = Elem[operand3, e, esize];

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

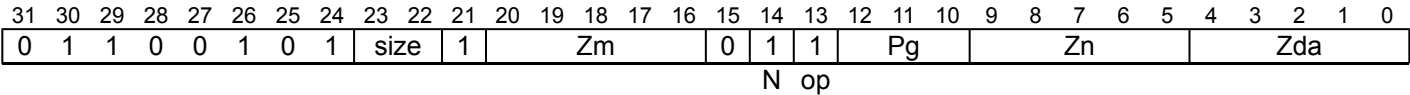
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FNMLS

Floating-point negated fused multiply-subtract vectors (predicated), writing addend [Zda = -Zda + Zn \* Zm]

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third source (addend) vector without intermediate rounding. Destructively place the negated results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.



FNMLS <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = TRUE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];

        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = Elem[operand3, e, esize];

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

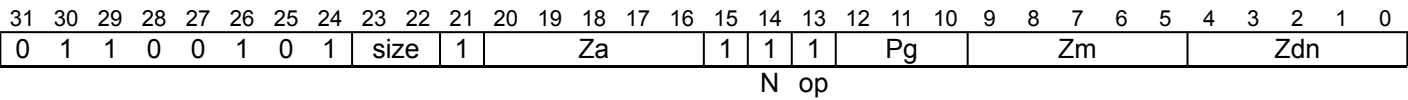
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FNMSB

Floating-point negated fused multiply-subtract vectors (predicated), writing multiplicand [ $Z_{dn} = -Z_a + Z_{dn} * Z_m$ ]

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third (addend) vector without intermediate rounding. Destructively place the negated results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.



FNMSB <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean op1_neg = FALSE;
boolean op3_neg = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Za> Is the name of the third source scalable vector register, encoded in the "Za" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];

        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRECPE

Floating-point reciprocal estimate (unpredicated)

Find the approximate reciprocal of each floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	1	1	1	0	0	0	1	1	0	0	Zn				Zd						

```
FRECPE <Zd>.<T>, <Zn>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRecipEstimate(element, FPCR[]);

Z[d, VL] = result;
```

FRECPS

Floating-point reciprocal step (unpredicated)

Multiply corresponding floating-point elements of the first and second source vectors, subtract the products from 2.0 without intermediate rounding and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction can be used to perform a single Newton-Raphson iteration for calculating the reciprocal of a vector of floating-point values.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	Zm				0	0	0	1	1	0	Zn				Zd						

FRECPS <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPREcipStepFused(element1, element2);

Z[d, VL] = result;
```

# FRECPX

Floating-point reciprocal exponent (predicated)

Invert the exponent and zero the fractional part of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

The result of this instruction can be used with FMULX to convert arbitrary elements in mathematical vector space to "unit vectors" or "direction vectors" of length 1.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	1	1	0	0	1	0	1	Pg				Zn				Zd					

**FRECPX** <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

## Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = FPRECPX(element, FPCR[]);

Z[d, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.



- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINT<r>

Floating-point round to integral value (predicated)

Round to an integral floating-point value with the specified rounding option from each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

<r>	Rounding Option
N	to nearest, with ties to even
A	to nearest, with ties away from zero
M	toward minus Infinity
P	toward plus Infinity
Z	toward zero
I	current FPCR rounding mode
X	current FPCR rounding mode, signalling inexact

It has encodings from 7 classes: [Current mode](#) , [Current mode signalling inexact](#) , [Nearest with ties to away](#) , [Nearest with ties to even](#) , [Toward zero](#) , [Toward minus infinity](#) and [Toward plus infinity](#)

Current mode

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	1	0	0	1	0	1	size		0	0	0	1	1	1	1	0	1	Pg													
																								Zn						Zd		

FRINTI <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Current mode signalling inexact

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	0	1	1	0	1	0	1	Pg			Zn			Zd							

FRINTX <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Nearest with ties to away

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	0	0	1	0	0	1	0	1	Pg			Zn					Zd				

**FRINTA** <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_TIEAWAY;
```

### Nearest with ties to even

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	0	0	0	0	0	1	0	1	Pg				Zn			Zd					

**FRINTN** <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_TIEEVEN;
```

### Toward zero

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	0	0	1	1	1	0	1	Pg													

**FRINTZ** <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

### Toward minus infinity

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	0	0	1	0	1	0	1	Pg													

**FRINTM** <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_NEGINF;
```

### Toward plus infinity

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	0	0	0	1	1	0	1	Pg													

FRINTP <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_POSINF;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

FRSQRTE

Floating-point reciprocal square root estimate (unpredicated)

Find the approximate reciprocal square root of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	1	1	1	1	0	0	1	1	0	0	Zn				Zd						

FRSQRTE <Zd>.<T>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRSqrtEstimate(element, FPCR[]);

Z[d, VL] = result;
```

FRSQRTS

Floating-point reciprocal square root step (unpredicated)

Multiply corresponding floating-point elements of the first and second source vectors, subtract the products from 3.0 and divide the results by 2.0 without any intermediate rounding and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.  
This instruction can be used to perform a single Newton-Raphson iteration for calculating the reciprocal square root of a vector of floating-point values.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	Zm				0	0	0	1	1	1	Zn				Zd						

FRSQRTS <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRSqrtStepFused(element1, element2);

Z[d, VL] = result;
```

FSCALE

Floating-point adjust exponent by vector (predicated)

Multiply the active floating-point elements of the first source vector by 2.0 to the power of the signed integer values in the corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	1	0	0	1	1	0	0	Pg			Zm				Zdn						

FSCALE <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ActivePredicateElement(mask, e, esize) then
    integer element2 = SInt(Elem[operand2, e, esize]);
    Elem[result, e, esize] = FPScale(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



FSQRT

Floating-point square root (predicated)

Calculate the square root of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	1	1	0	1	1	0	1	Pg													

FSQRT <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = FPSqrt(element, FPCR[]);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



FSUB (immediate)

Floating-point subtract immediate (predicated)

Subtract an immediate from each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	1	1	0	0	1	1	0	0	Pg	0	0	0	0	i1	Zdn							

FSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then FPPointFive('0', esize) else FPOne('0', esize);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in "i1":

i1	<const>
0	#0.5
1	#1.0

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = FPSub(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FSUB (vectors, predicated)

Floating-point subtract vectors (predicated)

Subtract active floating-point elements of the second source vector from corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	!= 00	0	0	0	0	0	0	1	1	0	0	Pg												
								size																Zm				Zdn			

**FSUB** <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

### Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPSub(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

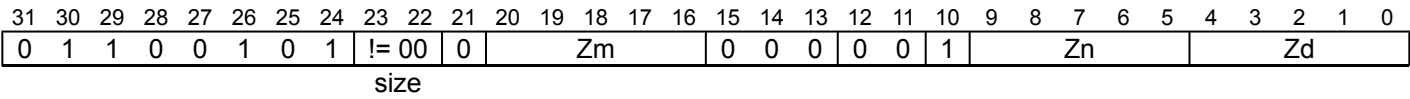
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FSUB (vectors, unpredicated)

Floating-point subtract vectors (unpredicated)

Subtract all floating-point elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.



```
FSUB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPSub(element1, element2, FPCR[]);

Z[d, VL] = result;
```

FSUBR (immediate)

Floating-point reversed subtract from immediate (predicated)

Reversed subtract from an immediate each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	1	1	0	1	1	1	0	0	Pg	0	0	0	0	i1	Zdn							

FSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then FPPointFive('0', esize) else FPOne('0', esize);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in "i1":

i1	<const>
0	#0.5
1	#1.0

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = FPSub(imm, element1, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```



## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FSUBR (vectors)

Floating-point reversed subtract vectors (predicated)

Reversed subtract active floating-point elements of the first source vector from corresponding floating-point elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	0	0	1	1	1	0	0	Pg	Zm				Zdn								

**FSUBR** <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

### Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ActivePredicateElement(mask, e, esize) then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPSub(element2, element1, FPCR[]);
  else
    Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FTMAD

Floating-point trigonometric multiply-add coefficient

The FTMAD instruction calculates the series terms for either SIN(x) or COS(x), where the argument x has been adjusted to be in the range  $-\pi/4 < x \leq \pi/4$ .

To calculate the series terms of SIN(x) and COS(x) the initial source operands of FTMAD should be zero in the first source vector and  $x^2$  in the second source vector. The FTMAD instruction is then executed eight times to calculate the sum of eight series terms, which gives a result of sufficient precision.

The FTMAD instruction multiplies each element of the first source vector by the absolute value of the corresponding element of the second source vector and performs a fused addition of each product with a value obtained from a table of hard-wired coefficients, and places the results destructively in the first source vector.

The coefficients are different for SIN(x) and COS(x), and are selected by a combination of the sign bit in the second source element and an immediate index in the range 0 to 7.

Double-precision coefficient table for sin(x) (s2<63> == '0')

Index	Hexadecimal	Decimal	Exact Value
0	3ff0 0000 0000 0000	1.0	= 1/1!
1	bfc5 5555 5555 5543	-0.16666666666666661	> -1/3!
2	3f81 1111 1110 f30c	0.83333333333320002e-02	< 1/5!
3	bf2a 01a0 19b9 2fc6	-0.1984126982840213e-03	> -1/7!
4	3ec7 1de3 51f3 d22b	0.2755731329901505e-05	< 1/9!
5	be5a e5e2 b60f 7b91	-0.2505070584637887e-07	> -1/11!
6	3de5 d840 8868 552f	0.1589413637195215e-09	< 1/13!
7	0000 0000 0000 0000	0.0	> -1/15!

Double-precision coefficient table for cos(x) (s2<63> == '1')

Index	Hexadecimal	Decimal	Exact Value
0	3ff0 0000 0000 0000	1.0	= 1/0!
1	bfe0 0000 0000 0000	-0.50000000000000000	= -1/2!
2	3fa5 5555 5555 5536	0.4166666666666645e-01	< 1/4!
3	bf56 c16c 16c1 3a0b	-0.1388888888886111e-02	> -1/6!
4	3efa 01a0 19b1 e8d8	0.2480158728388683e-04	< 1/8!
5	be92 7e4f 7282 f468	-0.2755731309913950e-06	> -1/10!
6	3e21 ee96 d264 1b13	0.2087558253975872e-08	< 1/12!
7	bda8 f763 80fb b401	-0.1135338700720054e-10	> -1/14!

Single-precision coefficient table for sin(x) (s2<31> == '0')

Index	Hexadecimal	Decimal	Exact Value
0	3f80 0000	1.0	= 1/1!
1	be2a aaab	-1.666666716337e-01	> -1/3!
2	3c08 8886	8.333330973983e-03	< 1/5!
3	b950 08b9	-1.983967522392e-04	> -1/7!
4	3636 9d6d	2.721174723774e-06	< 1/9!
5	0000 0000	0.0	> -1/11!
6	0000 0000	0.0	< 1/13!
7	0000 0000	0.0	> -1/15!

Single-precision coefficient table for cos(x) (s2<31> == '1')

Index	Hexadecimal	Decimal	Exact Value
0	3f80 0000	1.0	= 1/0!
1	bf00 0000	-5.000000000000e-01	= -1/2!
2	3d2a aaa6	4.166664928198e-02	< 1/4!
3	bab6 0705	-1.388759003021e-03	> -1/6!
4	37cd 37cc	2.446388680255e-05	< 1/8!
5	0000 0000	0.0	> -1/10!
6	0000 0000	0.0	< 1/12!
7	0000 0000	0.0	> -1/14!

Half-precision coefficient table for sin(x) (s2<15> == '0')

Index	Hexadecimal	Decimal	Exact Value
0	3c00	1.0	= 1/1!
1	b155	-1.666666716337e-01	> -1/3!
2	2030	8.333330973983e-03	< 1/5!
3	0000	0.0	> -1/7!
4	0000	0.0	< 1/9!
5	0000	0.0	> -1/11!
6	0000	0.0	< 1/13!
7	0000	0.0	> -1/15!

Half-precision coefficient table for cos(x) (s2<15> == '1')

Index	Hexadecimal	Decimal	Exact Value
0	3c00	1.0	= 1/0!
1	b800	-5.000000000000e-01	= -1/2!
2	293a	4.166664928198e-02	< 1/4!
3	0000	0.0	> -1/6!
4	0000	0.0	< 1/8!
5	0000	0.0	> -1/10!
6	0000	0.0	< 1/12!
7	0000	0.0	> -1/14!

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	1	0	imm3	1	0	0	0	0	0	0	0					Zm					Zdn	

**FTMAD** <Zdn>.<T>, <Zdn>.<T>, <Zm>.<T>, #<imm>

```

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer imm = UInt(imm3);

```

### Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<imm> Is the unsigned immediate operand, in the range 0 to 7, encoded in the "imm3" field.

### Operation

```

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPTrigMAdd(imm, element1, element2, FPCR[]);

Z[dn, VL] = result;

```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FTSMUL

Floating-point trigonometric starting value

The FTSMUL instruction calculates the initial value for the FTMAD instruction. The instruction squares each element in the first source vector and then sets the sign bit to a copy of bit 0 of the corresponding element in the second source register, and places the results in the destination vector. This instruction is unpredicated.

To compute SIN(x) or COS(x) the instruction is executed with elements of the first source vector set to x, adjusted to be in the range  $-\pi/4 < x \leq \pi/4$ .

The elements of the second source vector hold the corresponding value of the quadrant Q number as an integer not a floating-point value. The value Q satisfies the relationship  $(2q-1) \times \pi/4 < x \leq (2q+1) \times \pi/4$ .

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0				Zm			0	0	0	0	1	1				Zn				Zd		

FTSMUL <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPTrigSMul(element1, element2, FPCR[]);

Z[d, VL] = result;
```

FTSSEL

Floating-point trigonometric select coefficient

The FTSSEL instruction selects the coefficient for the final multiplication in the polynomial series approximation. The instruction places the value 1.0 or a copy of the first source vector element in the destination element, depending on bit 0 of the quadrant number Q held in the corresponding element of the second source vector. The sign bit of the destination element is copied from bit 1 of the corresponding value of Q. This instruction is unpredicated. To compute SIN(x) or COS(x) the instruction is executed with elements of the first source vector set to x, adjusted to be in the range  $-\pi/4 < x \leq \pi/4$ . The elements of the second source vector hold the corresponding value of the quadrant Q number as an integer not a floating-point value. The value Q satisfies the relationship  $(2q-1) \times \pi/4 < x \leq (2q+1) \times \pi/4$ . This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	1	0	0	size	1	Zm						1	0	1	1	0	0	Zn						Zd				

FTSSEL <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPTrigSSel(element1, element2);

Z[d, VL] = result;
```



HISTCNT

Count matching elements in vector

This instruction compares each active 32 or 64-bit element of the first source vector with all active elements with an element number less than or equal to its own in the second source vector, and places the count of matching elements in the corresponding element of the destination vector. Inactive elements in the destination vector are set to zero. This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size	1			Zm				1	1	0		Pg				Zn					Zd		

HISTCNT <Zd>.<T>, <Pg>/z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Zd);
integer n = UInt(Zn);
integer m = UInt(Zm);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <T> Is the size specifier, encoded in "size<0>":

size<0>	<T>
0	S
1	D
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer count = 0;
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element1 = Elem[operand1, e, esize];
        for i = 0 to e
            if ActivePredicateElement(mask, i, esize) then
                bits(esize) element2 = Elem[operand2, i, esize];
                if element1 == element2 then
                    count = count + 1;
        Elem[result, e, esize] = count<esize-1:0>;

Z[d, VL] = result;
```



HISTSEG

Count matching elements in vector segments

This instruction compares each 8-bit byte element of the first source vector with all of the elements in the corresponding 128-bit segment of the second source vector and places the count of matching elements in the corresponding element of the destination vector. This instruction is unpredicated.  
This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size		1	Zm					1	0	1	0	0	0	Zn					Zd				

HISTSEG <Zd>.B, <Zn>.B, <Zm>.B

```
if !HaveSVE2() then UNDEFINED;
if size != '00' then UNDEFINED;
constant integer esize = 8;
integer d = UInt(Zd);
integer n = UInt(Zn);
integer m = UInt(Zm);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for b = 0 to segments-1
  for s = 0 to eltspersegment-1
    integer count = 0;
    integer e = eltspersegment * b + s;
    bits(esize) element1 = Elem[operand1, e, esize];
    for i = 0 to eltspersegment-1
      integer e2 = eltspersegment * b + i;
      bits(esize) element2 = Elem[operand2, e2, esize];
      if element1 == element2 then
        count = count + 1;
      Elem[result, e, esize] = count<esize-1:0>;

Z[d, VL] = result;
```

## INCB, INCD, INCH, INCW (scalar)

Increment scalar by multiple of predicate constraint element count

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 4 classes: [Byte](#) , [Doubleword](#) , [Halfword](#) and [Word](#)

### Byte

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	1	1	imm4				1	1	1	0	0	0	pattern				Rdn					
size<1>size<0>																					D										

INCB <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

### Doubleword

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	1	1	1	imm4				1	1	1	0	0	0	pattern					Rdn				
size<1>size<0>																					D										

INCD <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

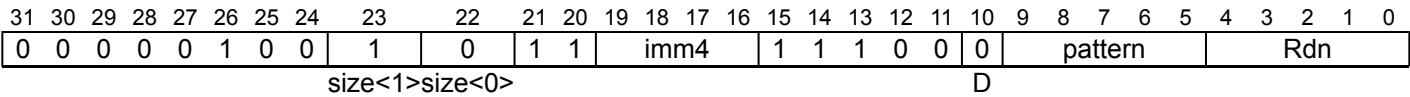
### Halfword

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	1	1	1	imm4				1	1	1	0	0	0	pattern				Rdn					
size<1>size<0>																					D										

INCH <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

Word



```
INCW <Xdn>{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

Assembler Symbols

- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

- <imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
constant integer VL = CurrentVL;
bits(64) operand1 = X[dn, 64];

X[dn, 64] = operand1 + (count * imm);
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



## INCD, INCH, INCW (vector)

Increment vector by multiple of predicate constraint element count

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 3 classes: [Doubleword](#) , [Halfword](#) and [Word](#)

### Doubleword

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	1	1	1	imm4				1	1	0	0	0	0	0	pattern				Zdn				
size<1>size<0>																					D										

```
INCD <Zdn>.D{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

### Halfword

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	1	1	1	imm4				1	1	0	0	0	0	0	pattern					Zdn			
size<1>size<0>																					D										

```
INCH <Zdn>.H{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

### Word

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	0	1	1	imm4				1	1	0	0	0	0	0	pattern					Zdn			
size<1>size<0>																					D										

```
INCW <Zdn>.S{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

### Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    Elem[result, e, esize] = Elem[operand1, e, esize] + (count * imm);

Z[dn, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

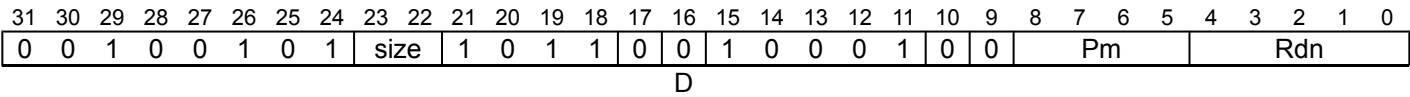
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



INCP (scalar)

Increment scalar by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to increment the scalar destination.



INCP <Xdn>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
```

Assembler Symbols

- <Xdn>Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Pm>Is the name of the source scalable predicate register, encoded in the "Pm" field.
- <T>Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) operand1 = X[dn, 64];
bits(PL) operand2 = P[m, PL];
integer count = 0;

for e = 0 to elements-1
    if ActivePredicateElement(operand2, e, esize) then
        count = count + 1;

X[dn, 64] = operand1 + count;
```

Operational information

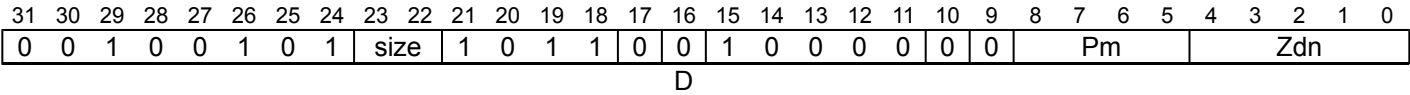
If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

INCP (vector)

Increment vector by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to increment all destination vector elements.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.



```
INCP <Zdn>.<T>, <Pm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) operand2 = P[m, PL];
bits(VL) result;
integer count = 0;

for e = 0 to elements-1
    if ActivePredicateElement(operand2, e, esize) then
        count = count + 1;

for e = 0 to elements-1
    Elem[result, e, esize] = Elem[operand1, e, esize] + count;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

INDEX (immediate, scalar)

Create index starting from immediate and incremented by general-purpose register

Populates the destination vector by setting the first element to the first signed immediate integer operand and monotonically incrementing the value by the second signed scalar integer operand for each subsequent element. The scalar source operand is a general-purpose register in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	0	0	0	1	0	0	size	1	Rm						0	1	0	0	1	0	imm5						Zd					

INDEX <Zd>.<T>, #<imm>, <R><m>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Rm);
integer d = UInt(Zd);
integer imm = SInt(imm5);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm> Is the signed immediate operand, in the range -16 to 15, encoded in the "imm5" field.

<R> Is a width specifier, encoded in "size":

size	<R>
01	W
x0	W
11	X

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(esize) operand2 = X[m, esize];
integer element2 = SInt(operand2);
bits(VL) result;

for e = 0 to elements-1
    integer index = imm + e * element2;
    Elem[result, e, esize] = index<size-1:0>;

Z[d, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

INDEX (immediates)

Create index starting from and incremented by immediate

Populates the destination vector by setting the first element to the first signed immediate integer operand and monotonically incrementing the value by the second signed immediate integer operand for each subsequent element. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1	imm5b						0	1	0	0	0	0	imm5					Zd				

INDEX <Zd>.<T>, #<imm1>, #<imm2>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer d = UInt(Zd);
integer imm1 = SInt(imm5);
integer imm2 = SInt(imm5b);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm1> Is the first signed immediate operand, in the range -16 to 15, encoded in the "imm5" field.

<imm2> Is the second signed immediate operand, in the range -16 to 15, encoded in the "imm5b" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) result;

for e = 0 to elements-1
    integer index = imm1 + e * imm2;
    Elem[result, e, esize] = index<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

INDEX (scalar, immediate)

Create index starting from general-purpose register and incremented by immediate

Populates the destination vector by setting the first element to the first signed scalar integer operand and monotonically incrementing the value by the second signed immediate integer operand for each subsequent element. The scalar source operand is a general-purpose register in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1	imm5					0	1	0	0	0	1	Rn					Zd					

```
INDEX <Zd>.<T>, <R><n>, #<imm>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer d = UInt(Zd);
integer imm = SInt(imm5);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in "size":

size	<R>
01	W
x0	W
11	X

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<imm> Is the signed immediate operand, in the range -16 to 15, encoded in the "imm5" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(esize) operand1 = X[n, esize];
integer element1 = SInt(operand1);
bits(VL) result;

for e = 0 to elements-1
    integer index = element1 + e * imm;
    Elem[result, e, esize] = index<esize-1:0>;

Z[d, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



INDEX (scalars)

Create index starting from and incremented by general-purpose register

Populates the destination vector by setting the first element to the first signed scalar integer operand and monotonically incrementing the value by the second signed scalar integer operand for each subsequent element. The scalar source operands are general-purpose registers in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	0	0	0	1	0	0	size	1	Rm						0	1	0	0	1	1	Rn						Zd					

INDEX <Zd>.<T>, <R><n>, <R><m>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in "size":

size	<R>
01	W
x0	W
11	X

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(esize) operand1 = X[n, esize];
integer element1 = SInt(operand1);
bits(esize) operand2 = X[m, esize];
integer element2 = SInt(operand2);
bits(VL) result;

for e = 0 to elements-1
    integer index = element1 + e * element2;
    Elem[result, e, esize] = index<esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

INSR (scalar)

Insert general-purpose register in shifted vector

Shift the destination vector left by one element, and then place a copy of the least-significant bits of the general-purpose register in element 0 of the destination vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	1	0	1	size	1	0	0	1	0	0	0	0	1	1	1	0	Rm						Zdn					

```
INSR <Zdn>.<T>, <R><m>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer m = UInt(Rm);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in "size":

size	<R>
01	W
x0	W
11	X

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(VL) dest = Z[dn, VL];
bits(esome) src = X[m, esize];
Z[dn, VL] = dest<(VL-esize)-1:0> : src;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

INSR (SIMD&FP scalar)

Insert SIMD&FP scalar register in shifted vector

Shift the destination vector left by one element, and then place a copy of the SIMD&FP scalar register in element 0 of the destination vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	1	0	1	0	0	0	0	1	1	1	0	Vm				Zdn						

```
INSR <Zdn>.<T>, <V><m>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer m = UInt(Vm);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<V> Is a width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	D

<m> Is the number [0-31] of the source SIMD&FP register, encoded in the "Vm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(VL) dest = Z[dn, VL];
bits(esize) src = V[m, esize];
Z[dn, VL] = dest<(VL-esize)-1:0> : src;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

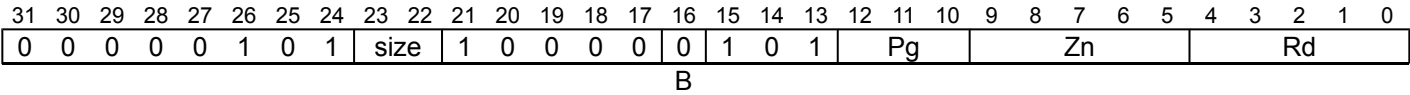
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LASTA (scalar)

Extract element after last to general-purpose register

If there is an active element then extract the element after the last active element modulo the number of elements from the final source vector register. If there are no active elements, extract element zero. Then zero-extend and place the extracted element in the destination general-purpose register.



```
LASTA <R><d>, <Pg>, <Zn>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = if esize < 64 then 32 else 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Rd);
boolean isBefore = FALSE;
```

Assembler Symbols

<R> Is a width specifier, encoded in “size”:

size	<R>
01	W
x0	W
11	X

<d> Is the number [0-30] of the destination general-purpose register or the name ZR (31), encoded in the "Rd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(PL) mask = P[g, PL];  
bits(VL) operand = Z[n, VL];  
bits(rsize) result;  
integer last = LastActiveElement(mask, esize);  
  
if isBefore then  
    if last < 0 then last = elements - 1;  
else  
    last = last + 1;  
    if last >= elements then last = 0;  
result = ZeroExtend(Elem[operand, last, esize], rsize);  
  
X[d, rsize] = result;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

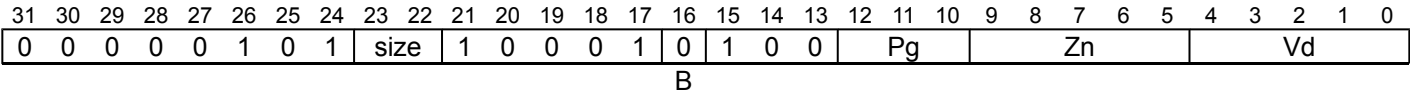
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



LASTA (SIMD&FP scalar)

Extract element after last to SIMD&FP scalar register

If there is an active element then extract the element after the last active element modulo the number of elements from the final source vector register. If there are no active elements, extract element zero. Then place the extracted element in the destination SIMD&FP scalar register.



```
LASTA <V><d>, <Pg>, <Zn>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean isBefore = FALSE;
```

Assembler Symbols

<V> Is a width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = Z[n, VL];
integer last = LastActiveElement(mask, esize);

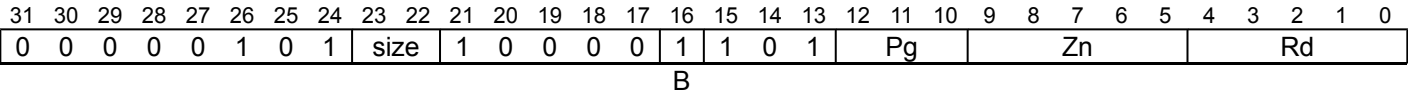
if isBefore then
    if last < 0 then last = elements - 1;
else
    last = last + 1;
    if last >= elements then last = 0;
V[d, esize] = Elem[operand, last, esize];
```



LASTB (scalar)

Extract last element to general-purpose register

If there is an active element then extract the last active element from the final source vector register. If there are no active elements, extract the highest-numbered element. Then zero-extend and place the extracted element in the destination general-purpose register.



```
LASTB <R><d>, <Pg>, <Zn>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = if esize < 64 then 32 else 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Rd);
boolean isBefore = TRUE;
```

Assembler Symbols

<R> Is a width specifier, encoded in “size”:

size	<R>
01	W
x0	W
11	X

<d> Is the number [0-30] of the destination general-purpose register or the name ZR (31), encoded in the "Rd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(PL) mask = P[g, PL];  
bits(VL) operand = Z[n, VL];  
bits(rsize) result;  
integer last = LastActiveElement(mask, esize);  
  
if isBefore then  
    if last < 0 then last = elements - 1;  
else  
    last = last + 1;  
    if last >= elements then last = 0;  
result = ZeroExtend(Elem[operand, last, esize], rsize);  
  
X[d, rsize] = result;
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

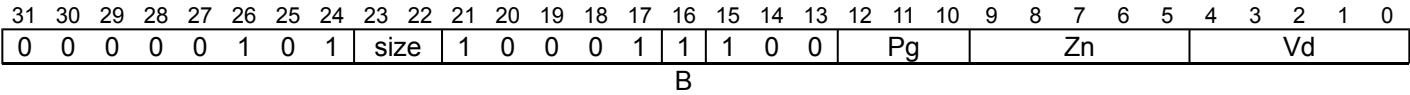
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LASTB (SIMD&FP scalar)

Extract last element to SIMD&FP scalar register

If there is an active element then extract the last active element from the final source vector register. If there are no active elements, extract the highest-numbered element. Then place the extracted element in the destination SIMD&FP register.



LASTB <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean isBefore = TRUE;
```

### Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = Z[n, VL];
integer last = LastActiveElement(mask, esize);

if isBefore then
    if last < 0 then last = elements - 1;
else
    last = last + 1;
    if last >= elements then last = 0;
V[d, esize] = Elem[operand, last, esize];
```



## LD1B (scalar plus immediate, consecutive registers)

Contiguous load of bytes to multiple consecutive vectors (immediate index)

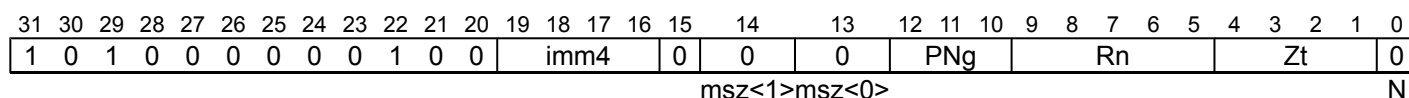
Contiguous load of unsigned bytes to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

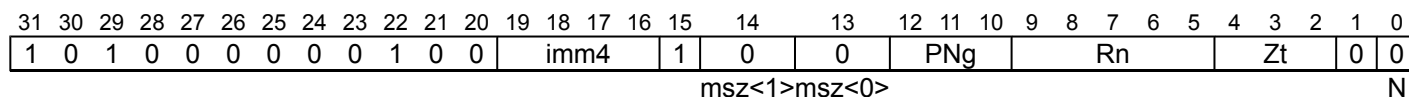


LD1B { <Zt1>.B-<Zt2>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 8;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SVE2p1)



LD1B { <Zt1>.B-<Zt4>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 8;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.
- For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## LD1B (scalar plus immediate, single register)

Contiguous load unsigned bytes to vector (immediate index)

Contiguous load of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 4 classes: [8-bit element](#) , [16-bit element](#) , [32-bit element](#) and [64-bit element](#)

### 8-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	0	0	0	0	imm4				1	0	1	Pg			Rn				Zt					
dtype<3:1>dtype<0>																															

**LD1B** { **<Zt>.B** }, **<Pg>/Z**, [**<Xn|SP>**{, #**<imm>**, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

### 16-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	0	0	1	0	imm4				1	0	1	Pg			Rn				Zt					
dtype<3:1>dtype<0>																															

**LD1B** { **<Zt>.H** }, **<Pg>/Z**, [**<Xn|SP>**{, #**<imm>**, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

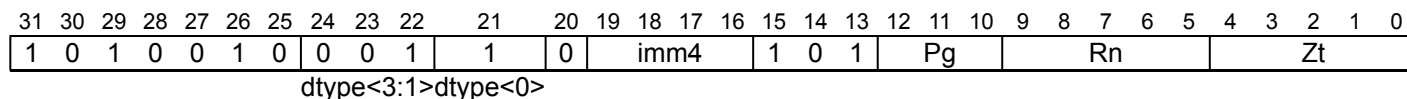
### 32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	0	1	0	0	imm4				1	0	1	Pg			Rn				Zt					
dtype<3:1>dtype<0>																															

**LD1B** { **<Zt>.S** }, **<Pg>/Z**, [**<Xn|SP>**{, #**<imm>**, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

## 64-bit element



**LD1B** { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

```

## Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

## Operation

```

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;

```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.



## LD1B (scalar plus scalar, consecutive registers)

Contiguous load of bytes to multiple consecutive vectors (scalar index)

Contiguous load of unsigned bytes to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	0					Rm	0	0	0			PNg								Zt		0
																msz<1>msz<0>														N	

LD1B { <Zt1>.B-<Zt2>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 8;
```

### Four registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	0					Rm	1	0	0			PNg								Zt	0	0
																msz<1>msz<0>														N	

LD1B { <Zt1>.B-<Zt4>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 8;
```

### Assembler Symbols

<Zt1>	For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.  For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1B (scalar plus scalar, single register)

Contiguous load unsigned bytes to vector (scalar index)

Contiguous load of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 4 classes: [8-bit element](#) , [16-bit element](#) , [32-bit element](#) and [64-bit element](#)

### 8-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	0	0		Rm				0	1	0	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

**LD1B** { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer msize = 8;
boolean unsigned = TRUE;
```

### 16-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	0	0	1	Rm				0	1	0	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

**LD1B** { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = TRUE;
```

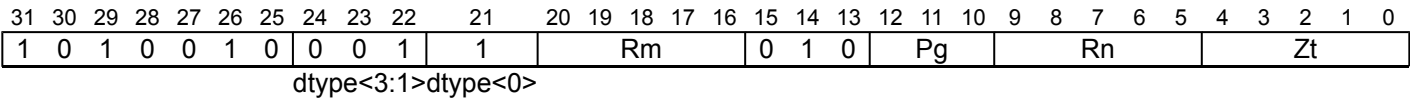
### 32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	0	1	0	Rm				0	1	0	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

LD1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
```

64-bit element



LD1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## LD1B (scalar plus vector)

Gather load unsigned bytes to vector (vector index)

Gather load of unsigned bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector. This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 3 classes: [32-bit unpacked unscaled offset](#) , [32-bit unscaled offset](#) and [64-bit unscaled offset](#)

### 32-bit unpacked unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	xs	0	Zm				0	1	0	Pg			Rn				Zt						
msz<1>msz<0>																U												ff			

**LD1B** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 32-bit unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	0	xs	0	Zm				0	1	0	Pg			Rn				Zt						
																U		ff													

**LD1B** { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 64-bit unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	1	0	Zm				1	1	0	Pg			Rn				Zt						
msz<1>msz<0>																U ff															

LD1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
constant integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
- <mod> Is the index extend and shift specifier, encoded in "xs":

xs	<mod>
0	UXTW
1	SXTW

### Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1B (vector plus immediate)

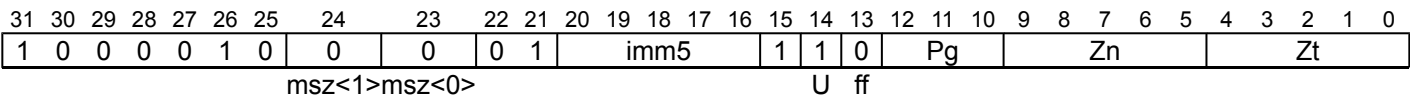
Gather load unsigned bytes to vector (immediate index)

Gather load of unsigned bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

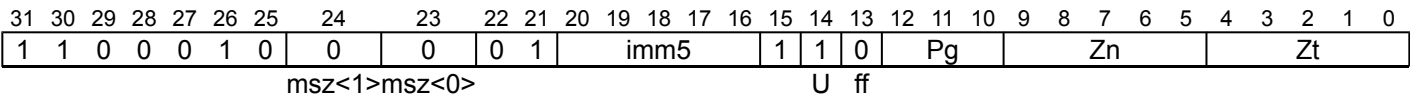
### 32-bit element



LD1B { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

### 64-bit element



LD1B { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1D (scalar plus immediate, consecutive registers)

Contiguous load of doublewords to multiple consecutive vectors (immediate index)

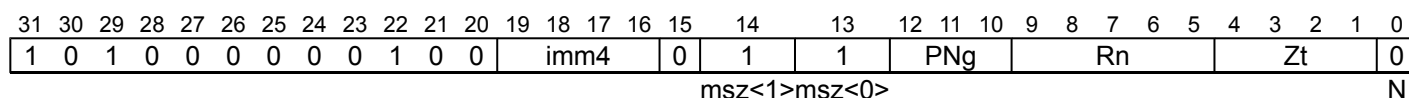
Contiguous load of unsigned doublewords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

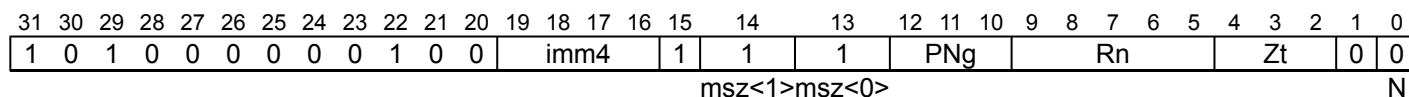


LD1D { <Zt1>.D-<Zt2>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SVE2p1)



LD1D { <Zt1>.D-<Zt4>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.  
For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

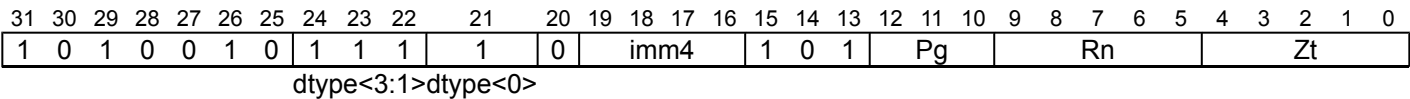
## LD1D (scalar plus immediate, single register)

Contiguous load unsigned doublewords to vector (immediate index)

Contiguous load of unsigned doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [SVE](#) and [SVE2](#)

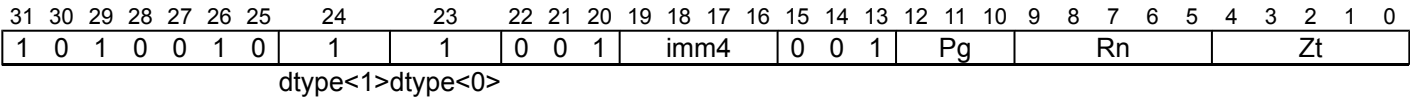
### SVE



LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

### SVE2 (FEAT\_SVE2p1)



LD1D { <Zt>.Q }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE2p1() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 128;
constant integer msize = 64;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.



## Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1D (scalar plus scalar, consecutive registers)

Contiguous load of doublewords to multiple consecutive vectors (scalar index)

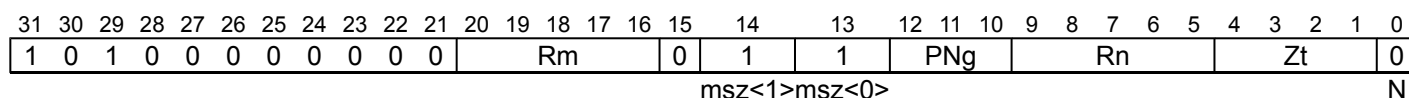
Contiguous load of unsigned doublewords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

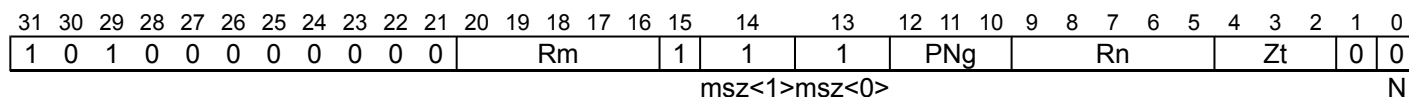


LD1D { <Zt1>.D-<Zt2>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 64;
```

### Four registers

(FEAT\_SVE2p1)



LD1D { <Zt1>.D-<Zt4>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 64;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.
- For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1D (scalar plus scalar, single register)

Contiguous load unsigned doublewords to vector (scalar index)

Contiguous load of unsigned doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [SVE](#) and [SVE2](#)

### SVE

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	1	1	1	1	Rm				0	1	0	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
```

### SVE2

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	1	1	0	0	Rm				1	0	0	Pg			Rn				Zt						
dtype<1>dtype<0>																															

LD1D { <Zt>.Q }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE2p1() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 128;
constant integer msize = 64;
boolean unsigned = TRUE;
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1D (scalar plus vector)

Gather load doublewords to vector (vector index)

Gather load of doublewords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 4 classes: [32-bit unpacked scaled offset](#) , [32-bit unpacked unscaled offset](#) , [64-bit scaled offset](#) and [64-bit unscaled offset](#)

### 32-bit unpacked scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	1	1	xs	1	Zm				0	1	0	Pg			Rn				Zt						
																U				ff											

**LD1D** { [<Zt>.D](#) }, [<Pg>/Z](#), [[<Xn|SP>](#), [<Zm>.D](#), [<mod>](#) #3]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 3;
```

### 32-bit unpacked unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	1	1	xs	0	Zm				0	1	0	Pg				Rn				Zt					
msz<1>msz<0>																U ff															

**LD1D** { [<Zt>.D](#) }, [<Pg>/Z](#), [[<Xn|SP>](#), [<Zm>.D](#), [<mod>](#)]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

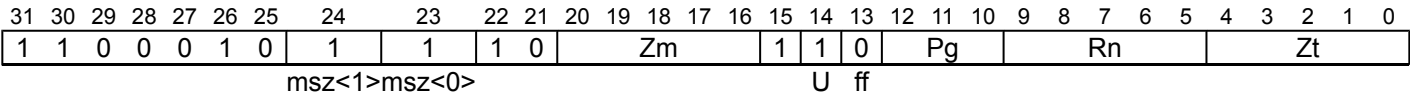
### 64-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	1	1	1	1	Zm				1	1	0	Pg			Rn				Zt						
																U				ff											

LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #3]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
constant integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 3;
```

64-bit unscaled offset



LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
constant integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
- <mod> Is the index extend and shift specifier, encoded in "xs":

xs	<mod>
0	UXTW
1	SXTW

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

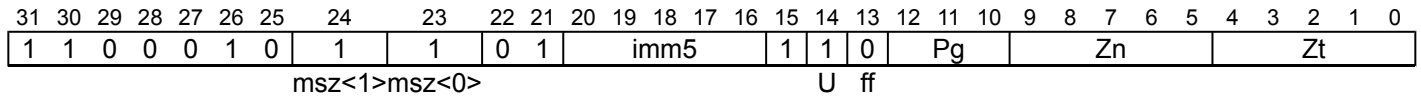


## LD1D (vector plus immediate)

Gather load doublewords to vector (immediate index)

Gather load of doublewords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.



**LD1D** { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

### Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1H (scalar plus immediate, consecutive registers)

Contiguous load of halfwords to multiple consecutive vectors (immediate index)

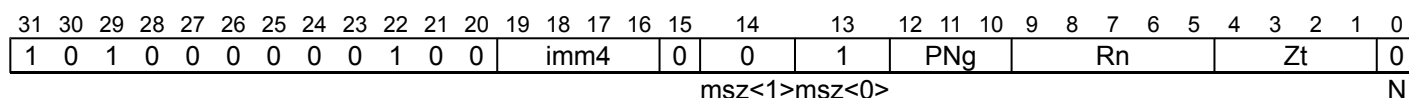
Contiguous load of unsigned halfwords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

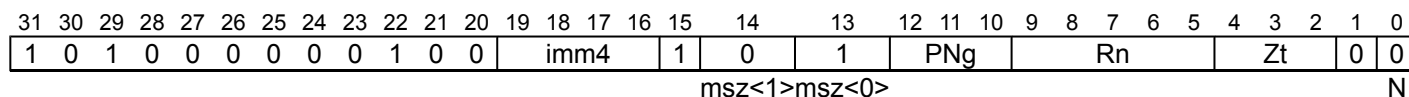


LD1H { <Zt1>.H-<Zt2>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 16;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SVE2p1)



LD1H { <Zt1>.H-<Zt4>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 16;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.
- For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1H (scalar plus immediate, single register)

Contiguous load unsigned halfwords to vector (immediate index)

Contiguous load of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: [16-bit element](#) , [32-bit element](#) and [64-bit element](#)

### 16-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	1	0	1	0	imm4				1	0	1	Pg			Rn				Zt					
dtype<3:1>dtype<0>																															

**LD1H** { **<Zt>.H** }, **<Pg>/Z**, [**<Xn|SP>**{, #**<imm>**, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

### 32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	1	1	0	0	imm4				1	0	1	Pg			Rn				Zt					
dtype<3:1>dtype<0>																															

**LD1H** { **<Zt>.S** }, **<Pg>/Z**, [**<Xn|SP>**{, #**<imm>**, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

### 64-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	1	1	1	0	imm4				1	0	1	Pg			Rn				Zt					
dtype<3:1>dtype<0>																															

**LD1H** { **<Zt>.D** }, **<Pg>/Z**, [**<Xn|SP>**{, #**<imm>**, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

## Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1H (scalar plus scalar, consecutive registers)

Contiguous load of halfwords to multiple consecutive vectors (scalar index)

Contiguous load of unsigned halfwords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm			0	0	1	PNg		Rn			Zt			0						
																msz<1>msz<0>														N	

LD1H { <Zt1>.H-<Zt2>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 16;
```

### Four registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	1	0	0	0	0	0	0	0	0	Rm			1	0	1	PNg		Rn			Zt			0	0						
																msz<1>msz<0>																N

LD1H { <Zt1>.H-<Zt4>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 16;
```

### Assembler Symbols

<Zt1>	For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.  For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## LD1H (scalar plus scalar, single register)

Contiguous load unsigned halfwords to vector (scalar index)

Contiguous load of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: [16-bit element](#) , [32-bit element](#) and [64-bit element](#)

### 16-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	1	0	1	Rm				0	1	0	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

LD1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 16;
boolean unsigned = TRUE;
```

### 32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	1	1	0	Rm				0	1	0	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

LD1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
```

### 64-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	1	1	1	Rm				0	1	0	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

```
LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
```

## Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1H (scalar plus vector)

Gather load unsigned halfwords to vector (vector index)

Gather load of unsigned halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 6 classes: [32-bit scaled offset](#) , [32-bit unpacked scaled offset](#) , [32-bit unpacked unscaled offset](#) , [32-bit unscaled offset](#) , [64-bit scaled offset](#) and [64-bit unscaled offset](#)

### 32-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	0	0	0	0	1	0	0	1	xs	1	Zm					0	1	0	Pg					Rn					Zt				
																U ff																	

**LD1H** { **<Zt>.S** }, **<Pg>/Z**, [**<Xn|SP>**, **<Zm>.S**, **<mod> #1**]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

### 32-bit unpacked scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	1	xs	1	Zm			0	1	0	Pg			Rn			Zt								
																U ff															

**LD1H** { **<Zt>.D** }, **<Pg>/Z**, [**<Xn|SP>**, **<Zm>.D**, **<mod> #1**]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

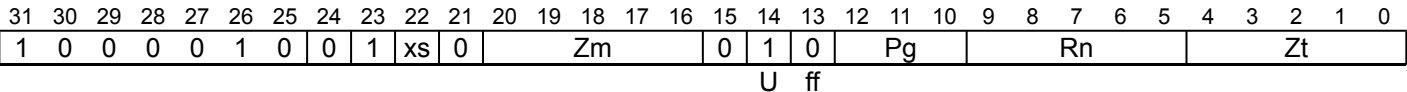
### 32-bit unpacked unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	1	xs	0			Zm		0	1	0		Pg					Rn						Zt	
msz<1>msz<0>																U ff															

LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

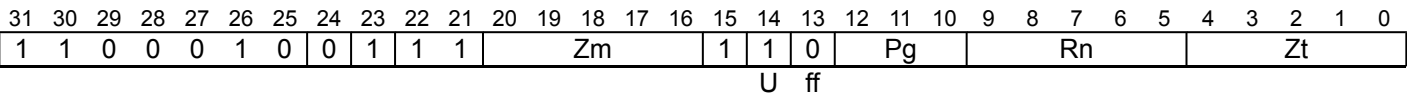
### 32-bit unscaled offset



LD1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

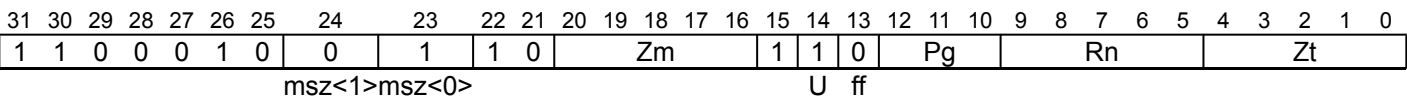
### 64-bit scaled offset



LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 1;
```

### 64-bit unscaled offset



LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
- <mod> Is the index extend and shift specifier, encoded in "xs":

xs	<mod>
0	UXTW
1	SXTW

### Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

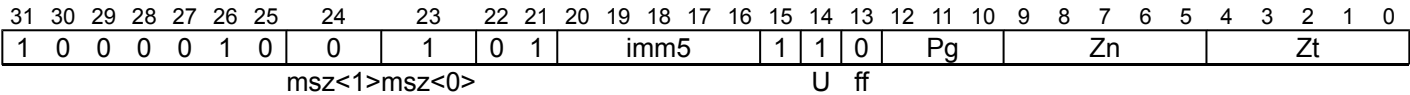
LD1H (vector plus immediate)

Gather load unsigned halfwords to vector (immediate index)

Gather load of unsigned halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.  
This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

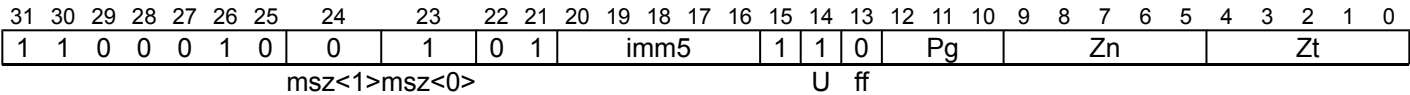
32-bit element



LD1H { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

64-bit element



LD1H { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



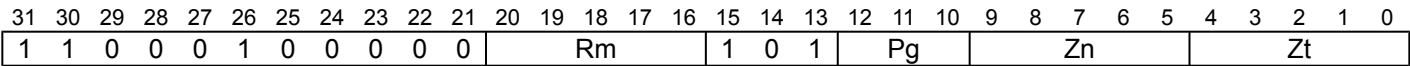
LD1Q

Gather load quadwords

Gather load of quadwords to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE2
(FEAT\_SVE2p1)



LD1Q { <Zt>.Q }, <Pg>/Z, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2p1() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
CheckNonStreamingSVEEnabled();
constant integer elements = VL DIV 128;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, 128) then
    base = Z[n, VL];
    offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, 128) then
        bits(64) addr = Elem[base, 2*e, 64] + offset;
        Elem[result, e, 128] = Mem[addr, 16, accdesc];
    else
        Elem[result, e, 128] = Zeros(128);

Z[t, VL] = result;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1RB

Load and broadcast unsigned byte to vector

Load a single unsigned byte from a memory address generated by a 64-bit scalar base address plus an immediate offset which is in the range 0 to 63.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 4 classes: [8-bit element](#) , [16-bit element](#) , [32-bit element](#) and [64-bit element](#)

### 8-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	0	1	imm6						1	0	0	Pg			Rn			Zt						
dtypeh<1>dtypeh<0>																dtypel<1>dtypel<0>															

LD1RB { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

### 16-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	0	1	imm6						1	0	1	Pg			Rn			Zt						
dtypeh<1>dtypeh<0>																dtypel<1>dtypel<0>															

LD1RB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

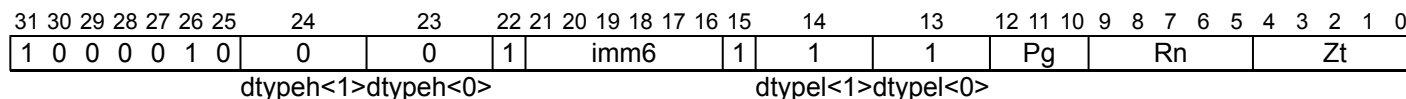
### 32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1 0 0 0 0 1 0							0	0	1	imm6						1	1	0	Pg			Rn			Zt						
dtypeh<1>dtypeh<0>																dtypel<1>dtypel<0>															

LD1RB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

## 64-bit element



LD1RB { <Zt>.D }, <Pg>/z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

## Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional unsigned immediate byte offset, in the range 0 to 63, defaulting to 0, encoded in the "imm6" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        bits(64) addr = base + offset * mbytes;
        data = Mem[addr, mbytes, accdesc];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

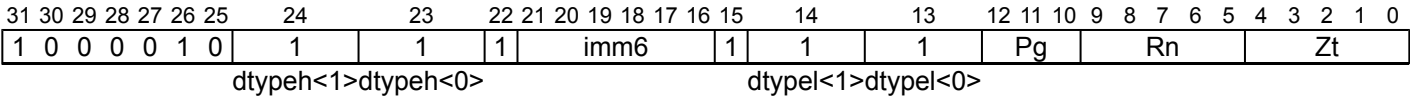


LD1RD

Load and broadcast doubleword to vector

Load a single doubleword from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 8 in the range 0 to 504.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.



```
LD1RD { <Zt>.D }, <Pg>/z, [<Xn|SP>{, #<imm>}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 504, defaulting to 0, encoded in the "imm6" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        bits(64) addr = base + offset * mbytes;
        data = Mem[addr, mbytes, accdesc];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1RH

Load and broadcast unsigned halfword to vector

Load a single unsigned halfword from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 2 in the range 0 to 126.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 3 classes: [16-bit element](#) , [32-bit element](#) and [64-bit element](#)

### 16-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	1	1	imm6						1	0	1	Pg			Rn			Zt						
dtypeh<1>dtypeh<0>																dtyel<1>dtyel<0>															

LD1RH { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

### 32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	1	1	imm6						1	1	0	Pg			Rn			Zt						
dtypeh<1>dtypeh<0>																dtyel<1>dtyel<0>															

LD1RH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

### 64-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	1	1	imm6						1	1	1	Pg			Rn			Zt						
dtypeh<1>dtypeh<0>																dtyel<1>dtyel<0>															

LD1RH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```



## Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 126, defaulting to 0, encoded in the "imm6" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        bits(64) addr = base + offset * mbytes;
        data = Mem[addr, mbytes, accdesc];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1ROB (scalar plus immediate)

Contiguous load and replicate thirty-two bytes (immediate index)

Load thirty-two contiguous bytes to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

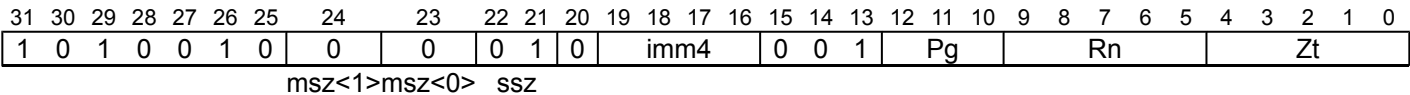
The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first thirty-two predicate elements are used and higher numbered predicate elements are ignored.

ID\_AA64ZFR0\_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE
(FEAT\_F64MM)



LD1ROB { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1ROB (scalar plus scalar)

Contiguous load and replicate thirty-two bytes (scalar index)

Load thirty-two contiguous bytes to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

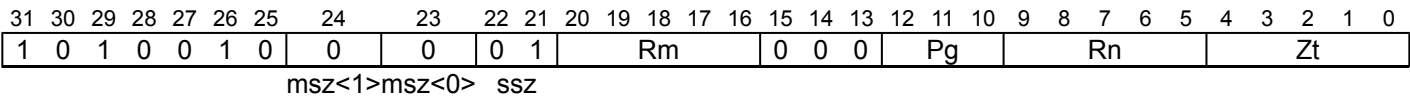
The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first thirty-two predicate elements are used and higher numbered predicate elements are ignored.

ID\_AA64ZFR0\_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE
(FEAT\_F64MM)



LD1ROB { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1ROD (scalar plus immediate)

Contiguous load and replicate four doublewords (immediate index)

Load four contiguous doublewords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

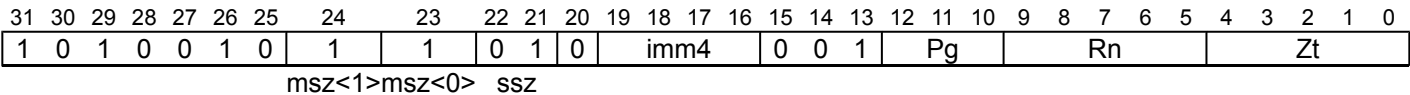
The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first four predicate elements are used and higher numbered predicate elements are ignored.

ID\_AA64ZFR0\_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE
(FEAT\_F64MM)



LD1ROD { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1ROD (scalar plus scalar)

Contiguous load and replicate four doublewords (scalar index)

Load four contiguous doublewords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 8 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

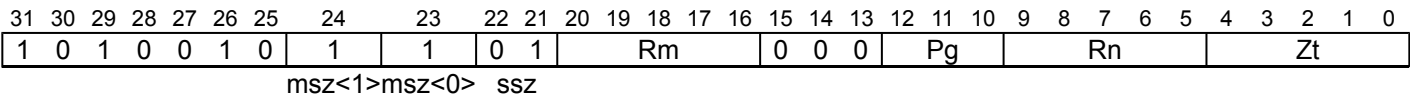
The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first four predicate elements are used and higher numbered predicate elements are ignored.

ID\_AA64ZFR0\_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE
(FEAT\_F64MM)



LD1ROD { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.



## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1ROH (scalar plus immediate)

Contiguous load and replicate sixteen halfwords (immediate index)

Load sixteen contiguous halfwords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

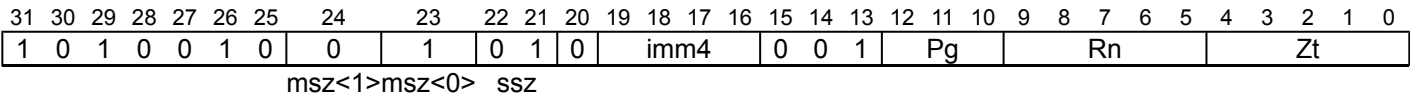
The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

ID\_AA64ZFR0\_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE
(FEAT\_F64MM)



LD1ROH { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1ROH (scalar plus scalar)

Contiguous load and replicate sixteen halfwords (scalar index)

Load sixteen contiguous halfwords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 2 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

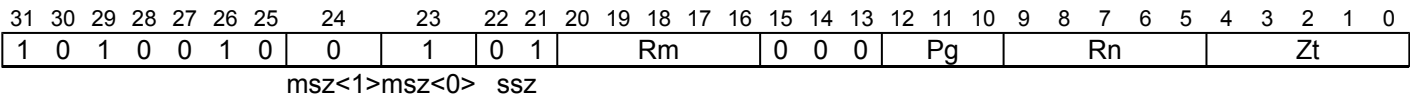
The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

ID\_AA64ZFR0\_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE
(FEAT\_F64MM)



LD1ROH { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1ROW (scalar plus immediate)

Contiguous load and replicate eight words (immediate index)

Load eight contiguous words to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

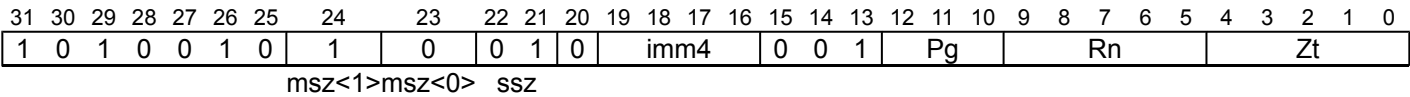
The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

ID\_AA64ZFR0\_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE  
(FEAT\_F64MM)



```
LD1ROW { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

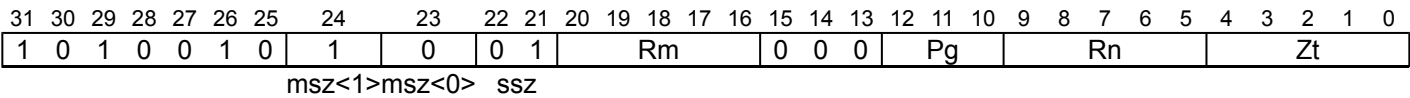
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1ROW (scalar plus scalar)

Contiguous load and replicate eight words (scalar index)

Load eight contiguous words to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 4 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero. Only the first eight predicate elements are used and higher numbered predicate elements are ignored. ID\_AA64ZFR0\_EL1.F64MM indicates whether this instruction is implemented. This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE
(FEAT\_F64MM)



LD1ROW { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.



## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

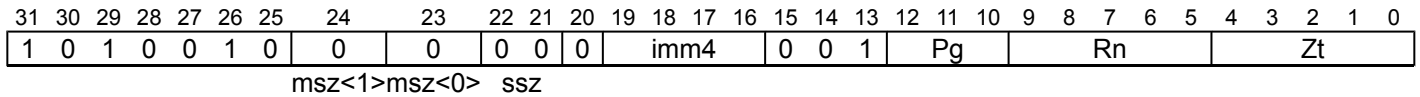
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1RQB (scalar plus immediate)

Contiguous load and replicate sixteen bytes (immediate index)

Load sixteen contiguous bytes to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.



```
LD1RQB { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (offset * 16) + (e * mbytes);
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = Replicate(result, VL DIV 128);
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

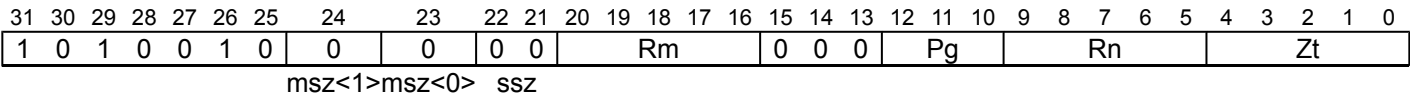
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1RQB (scalar plus scalar)

Contiguous load and replicate sixteen bytes (scalar index)

Load sixteen contiguous bytes to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is added to the base address.  
Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.



```
LD1RQB { <Zt>.B }, <Pg>/z, [<Xn|SP>, <Xm>]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = Replicate(result, VL DIV 128);
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

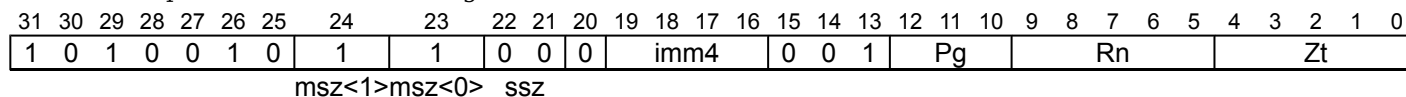
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1RQD (scalar plus immediate)

Contiguous load and replicate two doublewords (immediate index)

Load two contiguous doublewords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.



```
LD1RQD { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (offset * 16) + (e * mbytes);
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = Replicate(result, VL DIV 128);
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

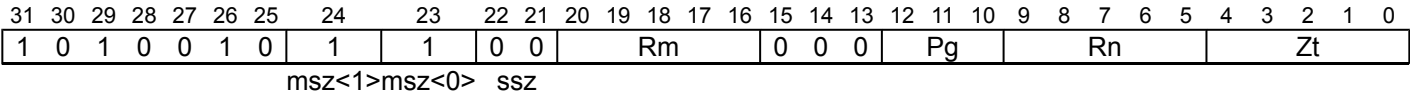
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1RQD (scalar plus scalar)

Contiguous load and replicate two doublewords (scalar index)

Load two contiguous doublewords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 8 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.



```
LD1RQD { <Zt>.D }, <Pg>/z, [<Xn|SP>, <Xm>, LSL #3]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = Replicate(result, VL DIV 128);
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

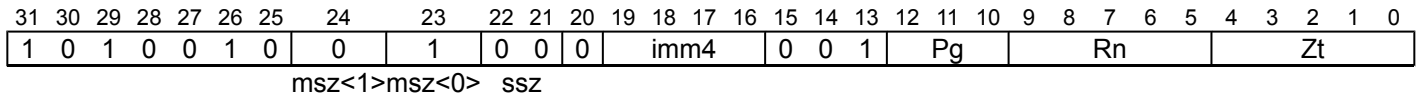
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1RQH (scalar plus immediate)

Contiguous load and replicate eight halfwords (immediate index)

Load eight contiguous halfwords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first eight predicate elements are used and higher numbered predicate elements are ignored.



```
LD1RQH { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (offset * 16) + (e * mbytes);
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = Replicate(result, VL DIV 128);
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

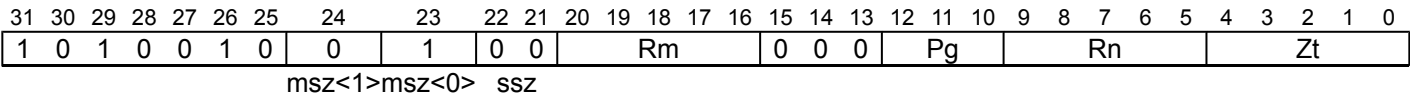
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1RQH (scalar plus scalar)

Contiguous load and replicate eight halfwords (scalar index)

Load eight contiguous halfwords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 2 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first eight predicate elements are used and higher numbered predicate elements are ignored.



```
LD1RQH { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = Replicate(result, VL DIV 128);
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

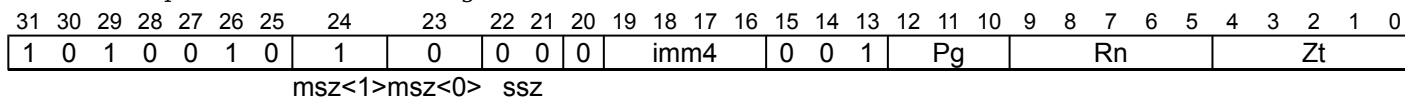
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1RQW (scalar plus immediate)

Contiguous load and replicate four words (immediate index)

Load four contiguous words to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first four predicate elements are used and higher numbered predicate elements are ignored.



```
LD1RQW { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (offset * 16) + (e * mbytes);
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = Replicate(result, VL DIV 128);
```

**Operational information**

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

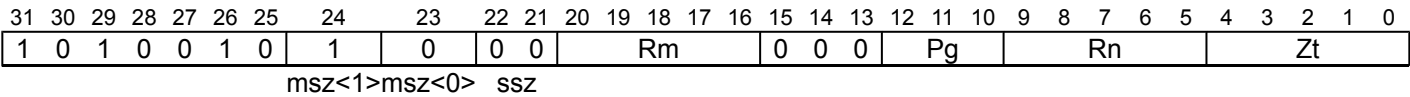
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1RQW (scalar plus scalar)

Contiguous load and replicate four words (scalar index)

Load four contiguous words to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 4 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first four predicate elements are used and higher numbered predicate elements are ignored.



```
LD1RQW { <Zt>.S }, <Pg>/z, [<Xn|SP>, <Xm>, LSL #2]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = Replicate(result, VL DIV 128);
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1RSB

Load and broadcast signed byte to vector

Load a single signed byte from a memory address generated by a 64-bit scalar base address plus an immediate offset which is in the range 0 to 63.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 3 classes: [16-bit element](#) , [32-bit element](#) and [64-bit element](#)

### 16-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1	1	1	imm6						1	1	0	Pg			Rn				Zt					
dtypeh<1>dtypeh<0>																dtytel<1>dtytel<0>															

**LD1RSB** { **<Zt>.H** }, **<Pg>/Z**, [ **<Xn|SP>**{, #**<imm>**} ]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

### 32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1	1	1	imm6						1	0	1	Pg			Rn				Zt					
dtypeh<1>dtypeh<0>																dtytel<1>dtytel<0>															

**LD1RSB** { **<Zt>.S** }, **<Pg>/Z**, [ **<Xn|SP>**{, #**<imm>**} ]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

### 64-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1 0 0 0 0 1 0							1	1	1	imm6						1	0	0	Pg			Rn				Zt					
dtypeh<1>dtypeh<0>																dtytel<1>dtytel<0>															

**LD1RSB** { **<Zt>.D** }, **<Pg>/Z**, [ **<Xn|SP>**{, #**<imm>**} ]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

## Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional unsigned immediate byte offset, in the range 0 to 63, defaulting to 0, encoded in the "imm6" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        bits(64) addr = base + offset * mbytes;
        data = Mem[addr, mbytes, accdesc];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

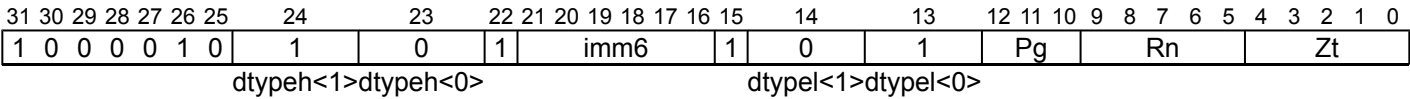
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1RSH

Load and broadcast signed halfword to vector

Load a single signed halfword from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 2 in the range 0 to 126.  
Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.  
It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

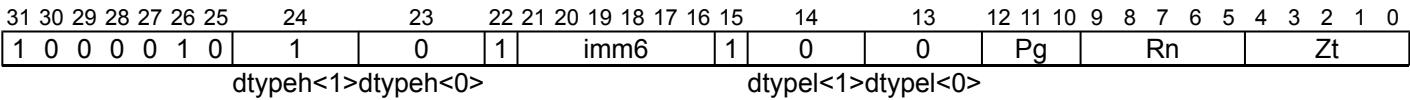
32-bit element



```
LD1RSH { <Zt>.S }, <Pg>/z, [<Xn|SP>{, #<imm>}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

64-bit element



```
LD1RSH { <Zt>.D }, <Pg>/z, [<Xn|SP>{, #<imm>}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 126, defaulting to 0, encoded in the "imm6" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        bits(64) addr = base + offset * mbytes;
        data = Mem[addr, mbytes, accdesc];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

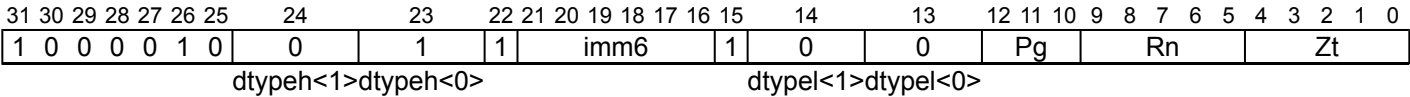
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1RSW

Load and broadcast signed word to vector

Load a single signed word from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 4 in the range 0 to 252.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.



```
LD1RSW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 252, defaulting to 0, encoded in the "imm6" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        bits(64) addr = base + offset * mbytes;
        data = Mem[addr, mbytes, accdesc];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LD1RW

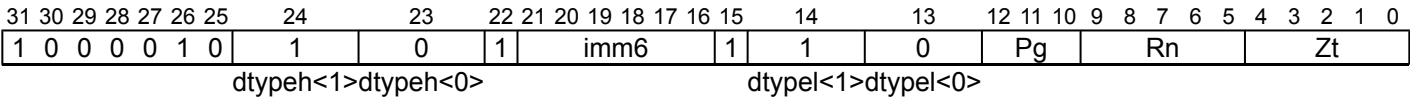
Load and broadcast unsigned word to vector

Load a single unsigned word from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 4 in the range 0 to 252.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

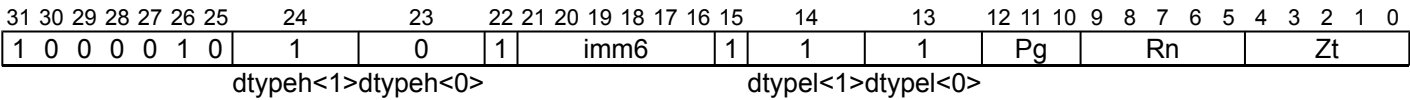
## 32-bit element



```
LD1RW { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

## 64-bit element



```
LD1RW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

## Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 252, defaulting to 0, encoded in the "imm6" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        bits(64) addr = base + offset * mbytes;
        data = Mem[addr, mbytes, accdesc];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1SB (scalar plus immediate)

Contiguous load signed bytes to vector (immediate index)

Contiguous load of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: [16-bit element](#) , [32-bit element](#) and [64-bit element](#)

### 16-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	1	1	1	0	0	imm4				1	0	1	Pg			Rn				Zt					
dtype<3:1>dtype<0>																															

LD1SB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

### 32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	1	1	0	1	0	imm4				1	0	1	Pg			Rn				Zt					
dtype<3:1>dtype<0>																															

LD1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

### 64-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	1	1	0	0	0	imm4				1	0	1	Pg			Rn				Zt					
dtype<3:1>dtype<0>																															

LD1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

## Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1SB (scalar plus scalar)

Contiguous load signed bytes to vector (scalar index)

Contiguous load of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: [16-bit element](#) , [32-bit element](#) and [64-bit element](#)

### 16-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	1	1	1	0	Rm				0	1	0	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

**LD1SB** { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = FALSE;
```

### 32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	1	1	0	1	Rm				0	1	0	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

**LD1SB** { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
```

### 64-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	1	1	0	0	Rm				0	1	0	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

```
LD1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
```

## Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1SB (scalar plus vector)

Gather load signed bytes to vector (vector index)

Gather load of signed bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector. This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 3 classes: [32-bit unpacked unscaled offset](#) , [32-bit unscaled offset](#) and [64-bit unscaled offset](#)

### 32-bit unpacked unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	xs	0	Zm				0	0	0	Pg			Rn				Zt						
msz<1>msz<0>																U ff															

**LD1SB** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
constant integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 32-bit unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	0	xs	0	Zm				0	0	0	Pg			Rn				Zt						
																U		ff													

**LD1SB** { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
constant integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 64-bit unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	1	0	Zm				1	0	0	Pg			Rn				Zt						
msz<1>msz<0>																U												ff			

LD1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
constant integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
- <mod> Is the index extend and shift specifier, encoded in "xs":

xs	<mod>
0	UXTW
1	SXTW

### Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



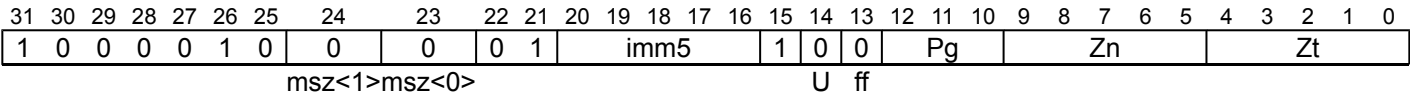
LD1SB (vector plus immediate)

Gather load signed bytes to vector (immediate index)

Gather load of signed bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.  
This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

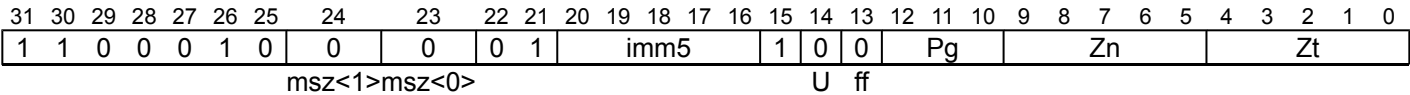
32-bit element



```
LD1SB { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

64-bit element



```
LD1SB { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

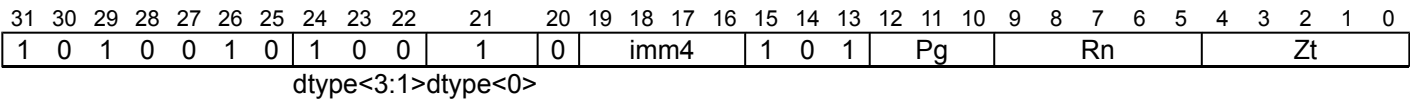
## LD1SH (scalar plus immediate)

Contiguous load signed halfwords to vector (immediate index)

Contiguous load of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

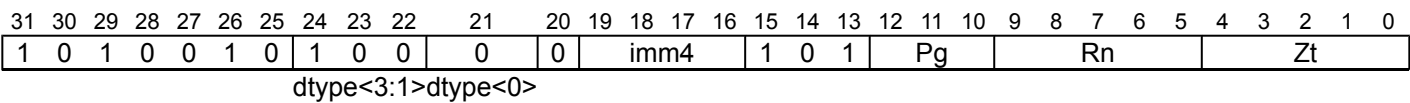
### 32-bit element



LD1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

### 64-bit element



LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

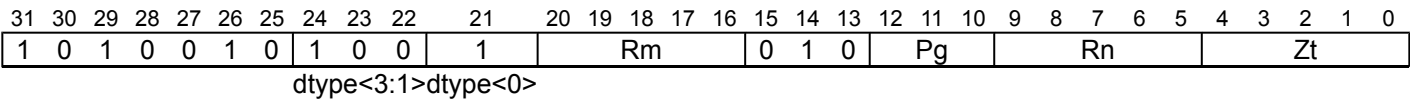
## LD1SH (scalar plus scalar)

Contiguous load signed halfwords to vector (scalar index)

Contiguous load of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

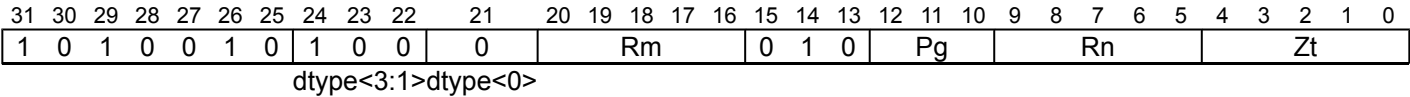
### 32-bit element



LD1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
```

### 64-bit element



LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1SH (scalar plus vector)

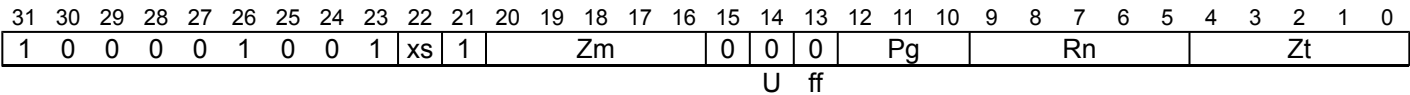
Gather load signed halfwords to vector (vector index)

Gather load of signed halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 6 classes: [32-bit scaled offset](#) , [32-bit unpacked scaled offset](#) , [32-bit unpacked unscaled offset](#) , [32-bit unscaled offset](#) , [64-bit scaled offset](#) and [64-bit unscaled offset](#)

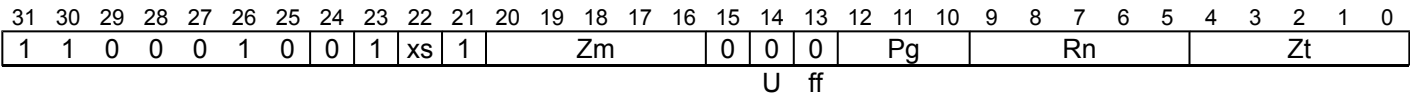
32-bit scaled offset



```
LD1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
constant integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

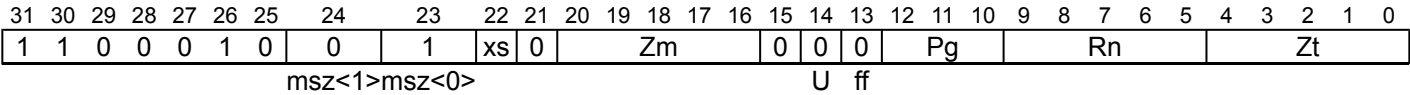
32-bit unpacked scaled offset



```
LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

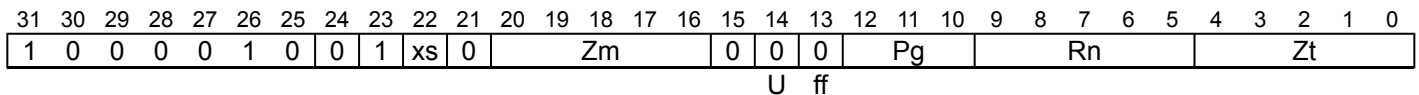
32-bit unpacked unscaled offset



LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

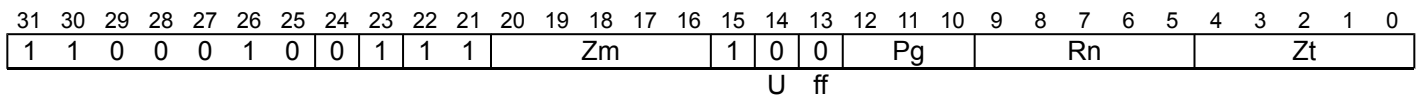
### 32-bit unscaled offset



LD1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
constant integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

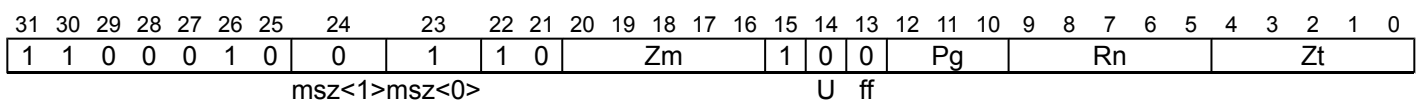
### 64-bit scaled offset



LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 1;
```

### 64-bit unscaled offset





LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
- <mod> Is the index extend and shift specifier, encoded in "xs":

xs	<mod>
0	UXTW
1	SXTW

### Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

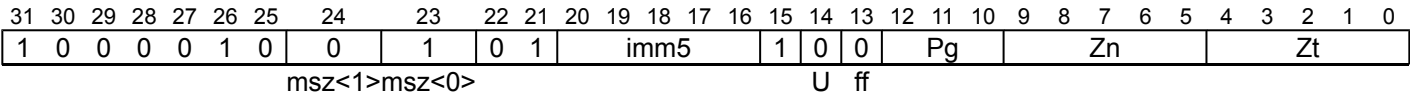
LD1SH (vector plus immediate)

Gather load signed halfwords to vector (immediate index)

Gather load of signed halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.  
This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

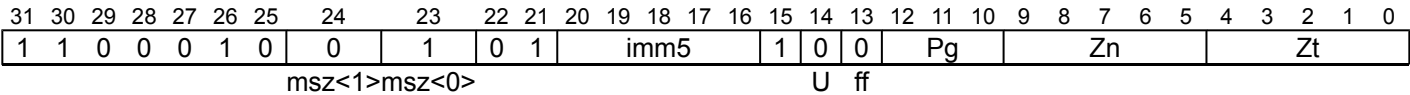
32-bit element



```
LD1SH { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

64-bit element



```
LD1SH { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

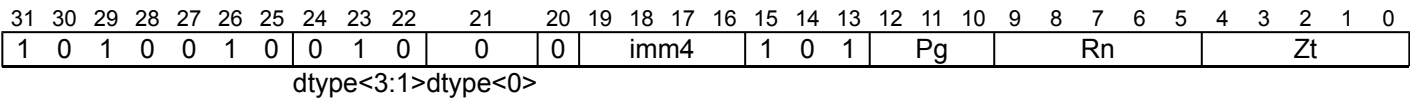
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1SW (scalar plus immediate)

Contiguous load signed words to vector (immediate index)

Contiguous load of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.



```
LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

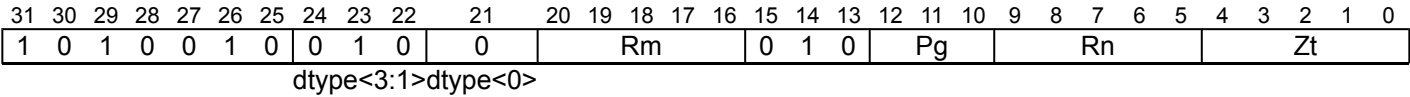
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1SW (scalar plus scalar)

Contiguous load signed words to vector (scalar index)

Contiguous load of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.



LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## LD1SW (scalar plus vector)

Gather load signed words to vector (vector index)

Gather load of signed words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 4 classes: [32-bit unpacked scaled offset](#) , [32-bit unpacked unscaled offset](#) , [64-bit scaled offset](#) and [64-bit unscaled offset](#)

### 32-bit unpacked scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	1	0	xs	1	Zm				0	0	0	Pg			Rn				Zt						
																U				ff											

**LD1SW** { **<Zt>.D** }, **<Pg>/Z**, [**<Xn|SP>**, **<Zm>.D**, **<mod> #2**]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

### 32-bit unpacked unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	1	0	xs	0	Zm			0	0	0	Pg			Rn			Zt								
msz<1>msz<0>																U ff															

**LD1SW** { **<Zt>.D** }, **<Pg>/Z**, [**<Xn|SP>**, **<Zm>.D**, **<mod>**]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

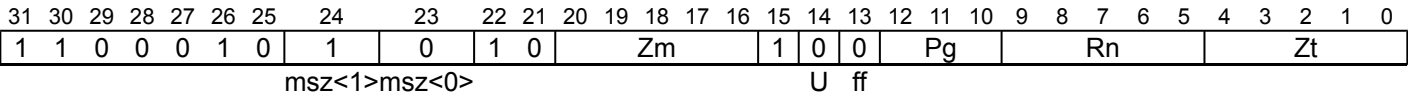
### 64-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	1	0	1	1	Zm				1	0	0	Pg			Rn				Zt						
																U				ff											

LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 2;
```

64-bit unscaled offset



LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
- <mod> Is the index extend and shift specifier, encoded in "xs":

xs	<mod>
0	UXTW
1	SXTW

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1SW (vector plus immediate)

Gather load signed words to vector (immediate index)

Gather load of signed words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	1	0	0	1	imm5					1	0	0	Pg			Zn				Zt					
msz<1>msz<0>																U ff															

**LD1SW** { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

### Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1W (scalar plus immediate, consecutive registers)

Contiguous load of words to multiple consecutive vectors (immediate index)

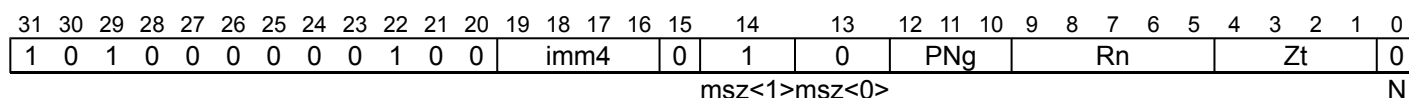
Contiguous load of unsigned words to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

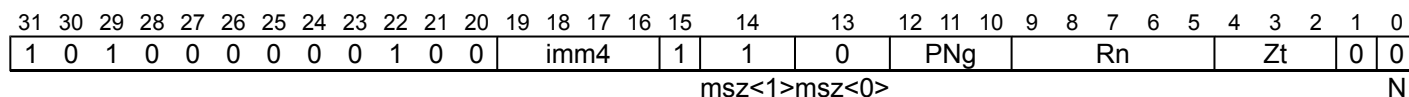


LD1W { <Zt1>.S-<Zt2>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 32;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SVE2p1)



LD1W { <Zt1>.S-<Zt4>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 32;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.  
For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1W (scalar plus immediate, single register)

Contiguous load unsigned words to vector (immediate index)

Contiguous load of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: [32-bit element](#) , [64-bit element](#) and [128-bit element](#)

### 32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	1	0	1	0	0	imm4			1	0	1	Pg			Rn			Zt							
dtype<3:1>dtype<0>																															

LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

### 64-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	1	0	1	1	0	imm4			1	0	1	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

### 128-bit element

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	1	0	0	0	1	imm4			0	0	1	Pg			Rn			Zt							
dtype<1>dtype<0>																															

LD1W { <Zt>.Q }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE2p1() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 128;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```



## Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

## Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1W (scalar plus scalar, consecutive registers)

Contiguous load of words to multiple consecutive vectors (scalar index)

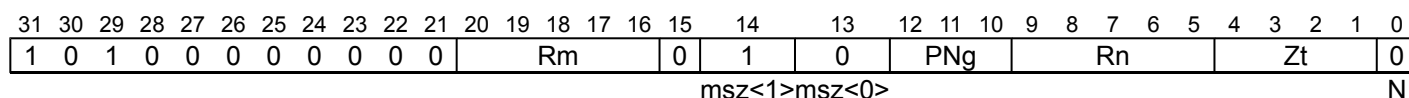
Contiguous load of unsigned words to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

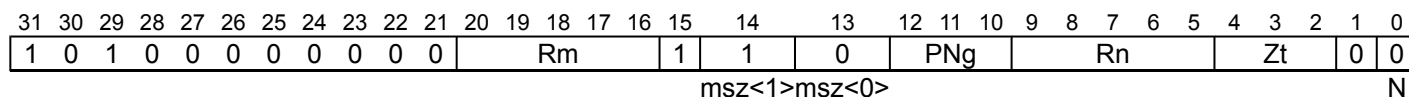


LD1W { <Zt1>.S-<Zt2>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 32;
```

### Four registers

(FEAT\_SVE2p1)



LD1W { <Zt1>.S-<Zt4>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 32;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.  
For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1W (scalar plus scalar, single register)

Contiguous load unsigned words to vector (scalar index)

Contiguous load of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: [32-bit element](#) , [64-bit element](#) and [128-bit element](#)

### 32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	1	0	1	0	Rm				0	1	0	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

**LD1W** { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
```

### 64-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	1	0	1	1	Rm				0	1	0	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

**LD1W** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
```

### 128-bit element

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	1	0	0	0	Rm				1	0	0	Pg			Rn				Zt						
dtype<1>dtype<0>																															

```
LD1W { <Zt>.Q }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

```
if !HaveSVE2p1() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 128;
constant integer msize = 32;
boolean unsigned = TRUE;
```

## Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1W (scalar plus vector)

Gather load unsigned words to vector (vector index)

Gather load of unsigned words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 6 classes: [32-bit scaled offset](#) , [32-bit unpacked scaled offset](#) , [32-bit unpacked unscaled offset](#) , [32-bit unscaled offset](#) , [64-bit scaled offset](#) and [64-bit unscaled offset](#)

### 32-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1	0	xs	1	Zm				0	1	0	Pg				Rn				Zt					
																U				ff											

**LD1W** { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

### 32-bit unpacked scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	1	0	xs	1	Zm				0	1	0	Pg				Rn				Zt					
																U				ff											

**LD1W** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

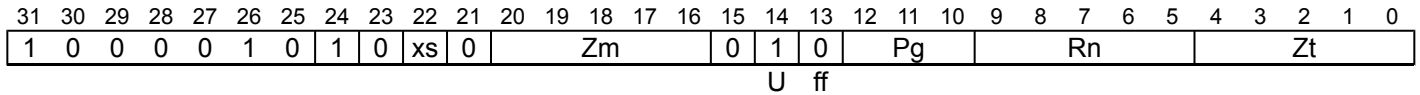
### 32-bit unpacked unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	1	0	xs	0			Zm			0	1	0		Pg				Rn				Zt			
msz<1>msz<0>																U				ff											

**LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]**

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

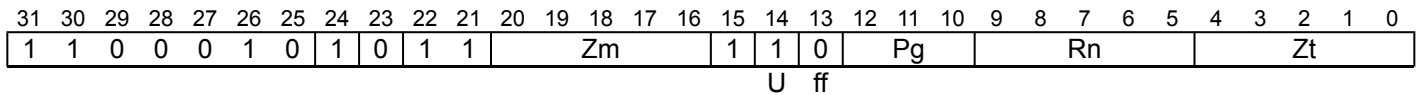
### 32-bit unscaled offset



**LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]**

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

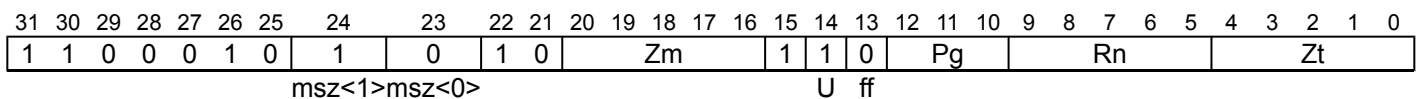
### 64-bit scaled offset



**LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]**

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 2;
```

### 64-bit unscaled offset



LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
- <mod> Is the index extend and shift specifier, encoded in "xs":

xs	<mod>
0	UXTW
1	SXTW

### Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```



## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

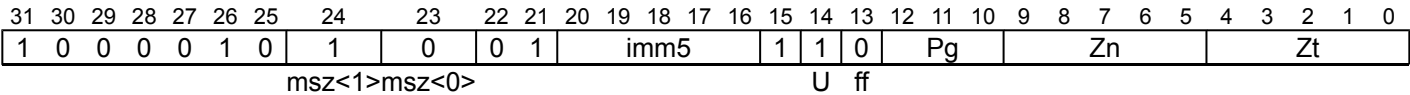
LD1W (vector plus immediate)

Gather load unsigned words to vector (immediate index)

Gather load of unsigned words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.  
This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

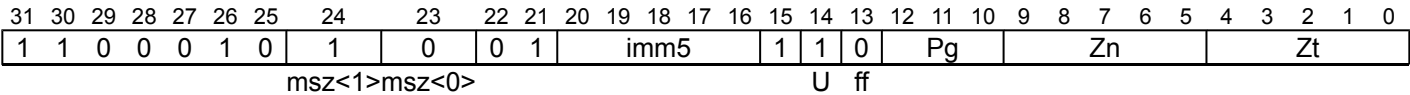
32-bit element



```
LD1W { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

64-bit element



```
LD1W { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

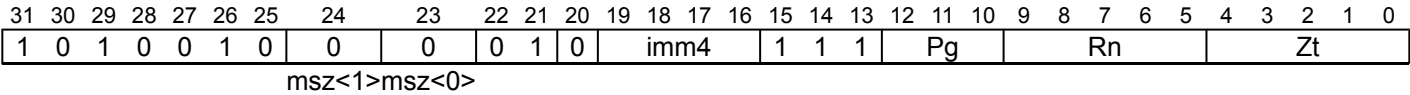
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD2B (scalar plus immediate)

Contiguous load two-byte structures to two vectors (immediate index)

Contiguous load two-byte structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.



```
LD2B { <Zt1>.B, <Zt2>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
constant integer nreg = 2;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

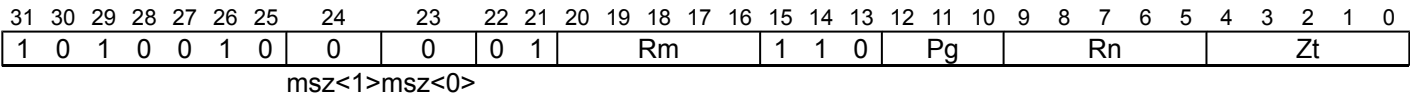
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD2B (scalar plus scalar)

Contiguous load two-byte structures to two vectors (scalar index)

Contiguous load two-byte structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.



```
LD2B { <Zt1>.B, <Zt2>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer nreg = 2;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

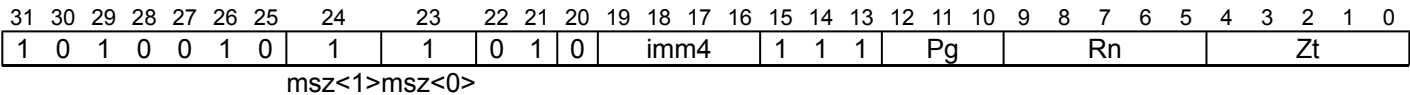
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD2D (scalar plus immediate)

Contiguous load two-doubleword structures to two vectors (immediate index)

Contiguous load two-doubleword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication, Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.



```
LD2D { <Zt1>.D, <Zt2>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
constant integer nreg = 2;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

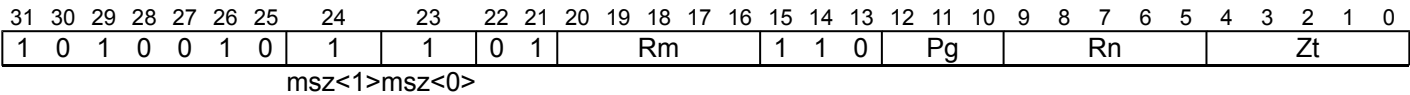
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD2D (scalar plus scalar)

Contiguous load two-doubleword structures to two vectors (scalar index)

Contiguous load two-doubleword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.



```
LD2D { <Zt1>.D, <Zt2>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer nreg = 2;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

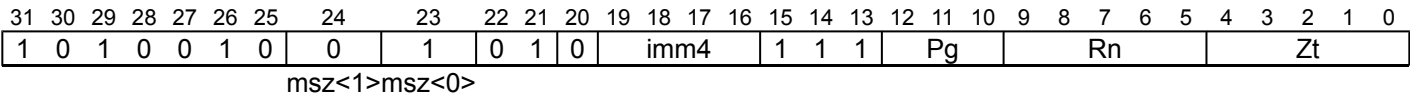
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD2H (scalar plus immediate)

Contiguous load two-halfword structures to two vectors (immediate index)

Contiguous load two-halfword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,  
Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.



```
LD2H { <Zt1>.H, <Zt2>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
constant integer nreg = 2;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

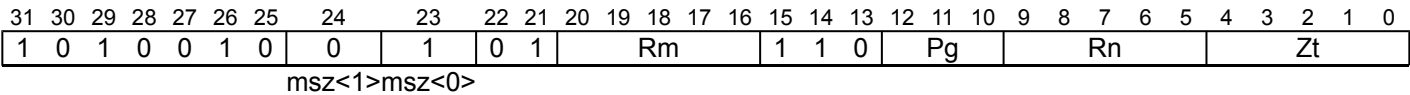
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD2H (scalar plus scalar)

Contiguous load two-halfword structures to two vectors (scalar index)

Contiguous load two-halfword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.



LD2H { <Zt1>.H, <Zt2>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer nreg = 2;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

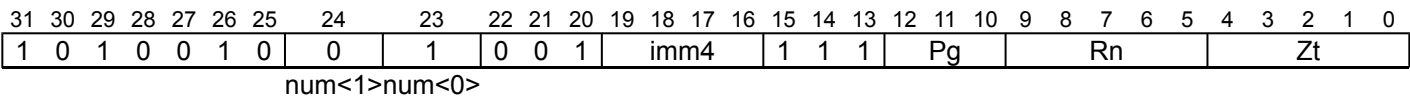
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD2Q (scalar plus immediate)

Contiguous load two-quadword structures to two vectors (immediate index)

Contiguous load two-quadword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,  
Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive quadwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

### SVE2 (FEAT\_SVE2p1)



LD2Q { <Zt1>.Q, <Zt2>.Q }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 128;
integer offset = SInt(imm4);
constant integer nreg = 2;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

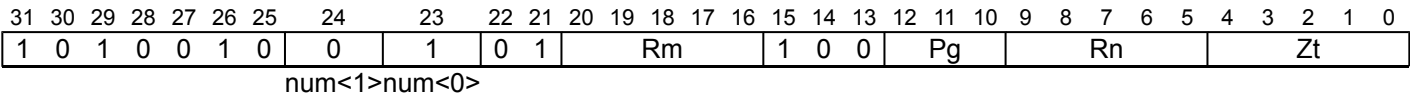
## LD2Q (scalar plus scalar)

Contiguous load two-quadword structures to two vectors (scalar index)

Contiguous load two-quadword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive quadwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

### SVE2 (FEAT\_SVE2p1)



LD2Q { <Zt1>.Q, <Zt2>.Q }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #4]

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 128;
constant integer nreg = 2;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

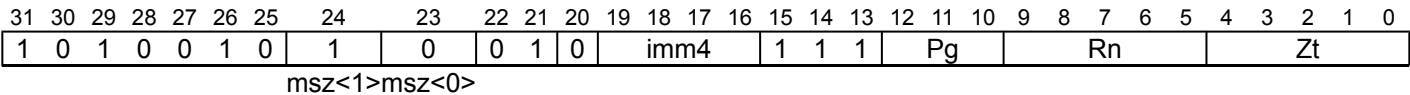
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD2W (scalar plus immediate)

Contiguous load two-word structures to two vectors (immediate index)

Contiguous load two-word structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.



```
LD2W { <Zt1>.S, <Zt2>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
constant integer nreg = 2;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

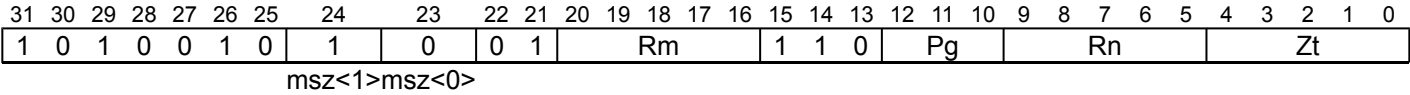
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD2W (scalar plus scalar)

Contiguous load two-word structures to two vectors (scalar index)

Contiguous load two-word structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.



LD2W { <Zt1>.S, <Zt2>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer nreg = 2;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

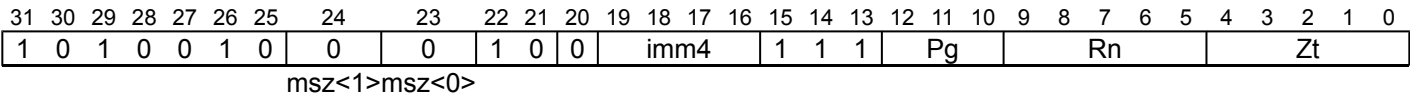
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD3B (scalar plus immediate)

Contiguous load three-byte structures to three vectors (immediate index)

Contiguous load three-byte structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication,  
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.



```
LD3B { <Zt1>.B, <Zt2>.B, <Zt3>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

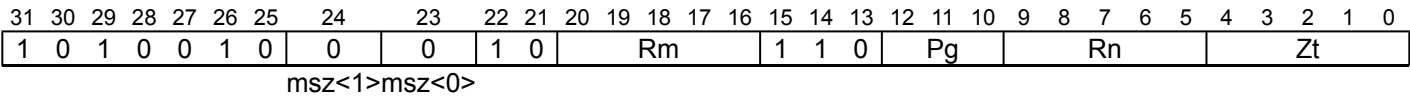
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD3B (scalar plus scalar)

Contiguous load three-byte structures to three vectors (scalar index)

Contiguous load three-byte structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.



LD3B { <Zt1>.B, <Zt2>.B, <Zt3>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer nreg = 3;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

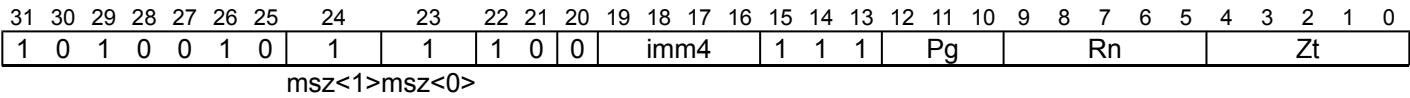
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD3D (scalar plus immediate)

Contiguous load three-doubleword structures to three vectors (immediate index)

Contiguous load three-doubleword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication, Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.



```
LD3D { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

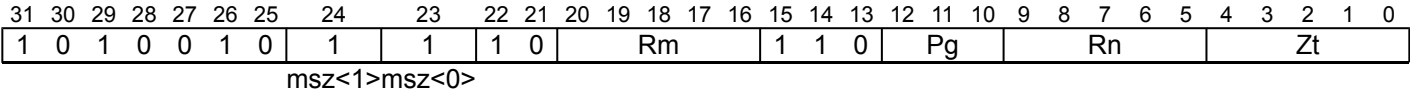
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD3D (scalar plus scalar)

Contiguous load three-doubleword structures to three vectors (scalar index)

Contiguous load three-doubleword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.



LD3D { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer nreg = 3;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

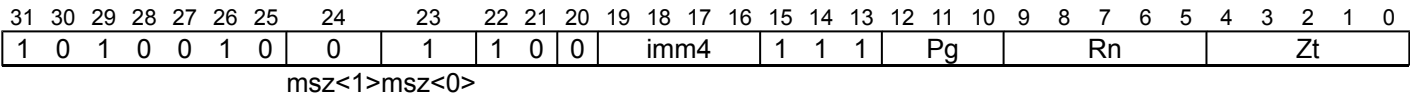
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD3H (scalar plus immediate)

Contiguous load three-halfword structures to three vectors (immediate index)

Contiguous load three-halfword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication, Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.



```
LD3H { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

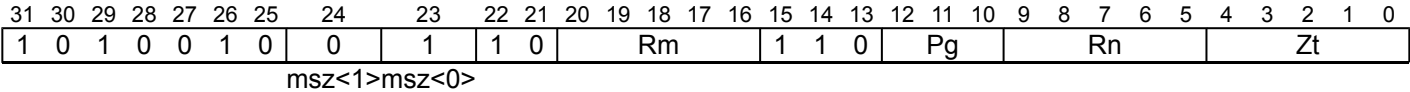
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD3H (scalar plus scalar)

Contiguous load three-halfword structures to three vectors (scalar index)

Contiguous load three-halfword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.



LD3H { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer nreg = 3;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

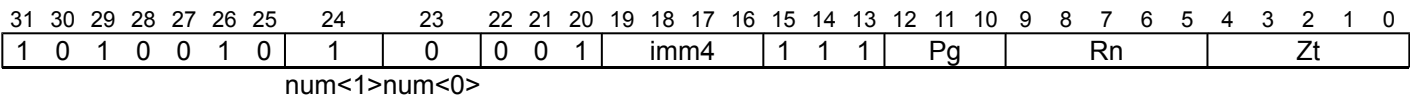
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD3Q (scalar plus immediate)

Contiguous load three-quadword structures to three vectors (immediate index)

Contiguous load three-quadword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication, Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive quadwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

### SVE2 (FEAT\_SVE2p1)



LD3Q { <Zt1>.Q, <Zt2>.Q, <Zt3>.Q }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 128;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

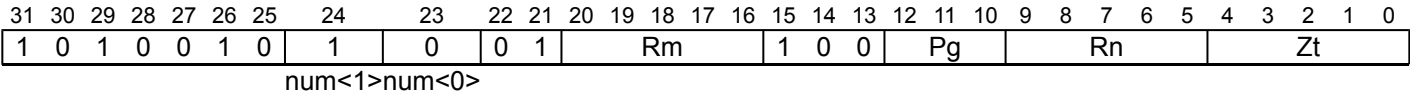
LD3Q (scalar plus scalar)

Contiguous load three-quadword structures to three vectors (scalar index)

Contiguous load three-quadword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive quadwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

SVE2  
(FEAT\_SVE2p1)



```
LD3Q { <Zt1>.Q, <Zt2>.Q, <Zt3>.Q }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #4]
```

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 128;
constant integer nreg = 3;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

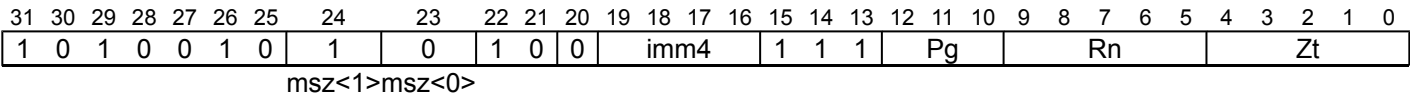
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD3W (scalar plus immediate)

Contiguous load three-word structures to three vectors (immediate index)

Contiguous load three-word structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication,  
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.



```
LD3W { <Zt1>.S, <Zt2>.S, <Zt3>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

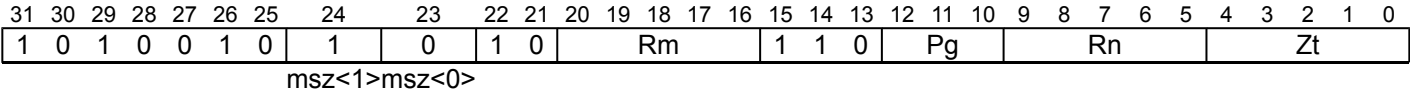
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD3W (scalar plus scalar)

Contiguous load three-word structures to three vectors (scalar index)

Contiguous load three-word structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.



LD3W { <Zt1>.S, <Zt2>.S, <Zt3>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer nreg = 3;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

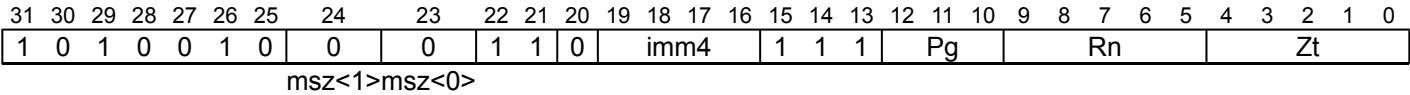
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD4B (scalar plus immediate)

Contiguous load four-byte structures to four vectors (immediate index)

Contiguous load four-byte structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.



```
LD4B { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
constant integer nreg = 4;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

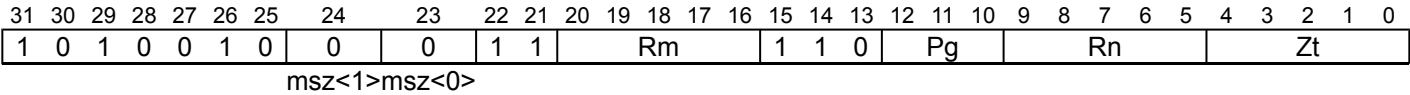
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD4B (scalar plus scalar)

Contiguous load four-byte structures to four vectors (scalar index)

Contiguous load four-byte structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.



**LD4B** { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer nreg = 4;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

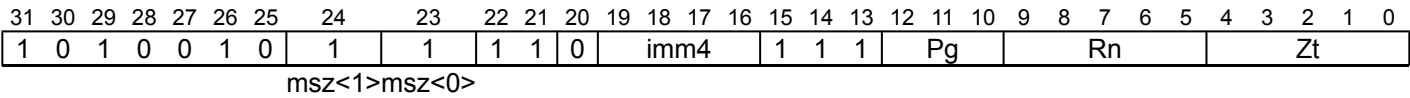
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD4D (scalar plus immediate)

Contiguous load four-doubleword structures to four vectors (immediate index)

Contiguous load four-doubleword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication, Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.



```
LD4D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
constant integer nreg = 4;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

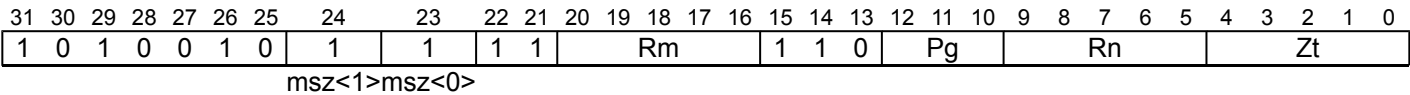
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD4D (scalar plus scalar)

Contiguous load four-doubleword structures to four vectors (scalar index)

Contiguous load four-doubleword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.



```
LD4D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer nreg = 4;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

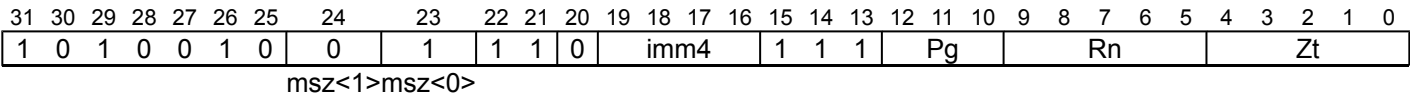
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD4H (scalar plus immediate)

Contiguous load four-halfword structures to four vectors (immediate index)

Contiguous load four-halfword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication,  
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.



```
LD4H { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
constant integer nreg = 4;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

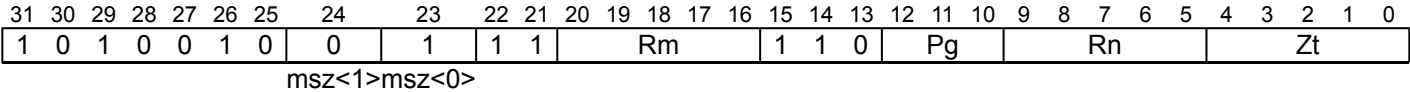
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD4H (scalar plus scalar)

Contiguous load four-halfword structures to four vectors (scalar index)

Contiguous load four-halfword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.



LD4H { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer nreg = 4;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

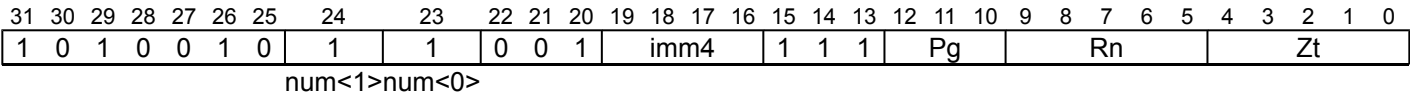
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD4Q (scalar plus immediate)

Contiguous load four-quadword structures to four vectors (immediate index)

Contiguous load four-quadword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication, Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive quadwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

SVE2
(FEAT\_SVE2p1)



LD4Q { <Zt1>.Q, <Zt2>.Q, <Zt3>.Q, <Zt4>.Q }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 128;
integer offset = SInt(imm4);
constant integer nreg = 4;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

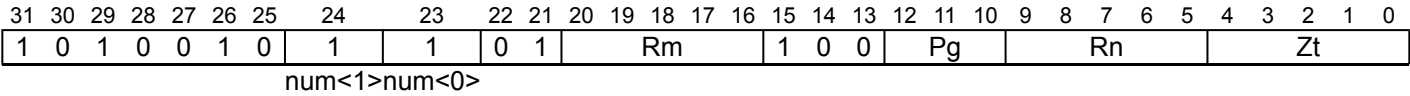
## LD4Q (scalar plus scalar)

Contiguous load four-quadword structures to four vectors (scalar index)

Contiguous load four-quadword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive quadwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

### SVE2 (FEAT\_SVE2p1)



LD4Q { <Zt1>.Q, <Zt2>.Q, <Zt3>.Q, <Zt4>.Q }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #4]

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 128;
constant integer nreg = 4;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

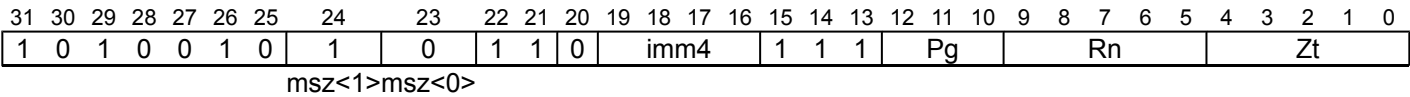
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD4W (scalar plus immediate)

Contiguous load four-word structures to four vectors (immediate index)

Contiguous load four-word structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.



```
LD4W { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
constant integer nreg = 4;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

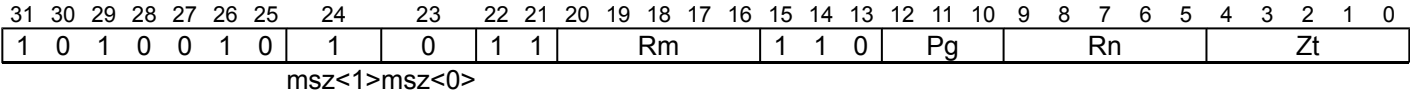
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD4W (scalar plus scalar)

Contiguous load four-word structures to four vectors (scalar index)

Contiguous load four-word structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.



```
LD4W { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer nreg = 4;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDFF1B (scalar plus scalar)

Contiguous load first-fault unsigned bytes to vector (scalar index)

Contiguous load with first-faulting behavior of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 4 classes: [8-bit element](#) , [16-bit element](#) , [32-bit element](#) and [64-bit element](#)

### 8-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	0	0		Rm				0	1	1	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

LDFF1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, <Xm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer msize = 8;
boolean unsigned = TRUE;
```

### 16-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	0	0	1	Rm				0	1	1	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

LDFF1B { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, <Xm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = TRUE;
```

### 32-bit element

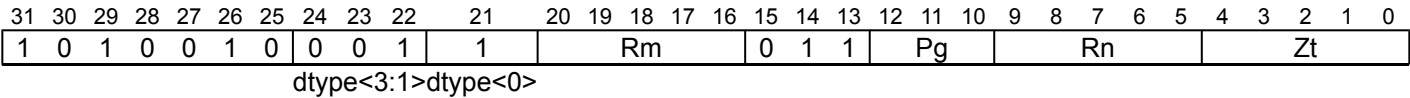
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	0	1	0	Rm				0	1	1	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															



```
LDFF1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
```

64-bit element



```
LDFF1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDFF1B (scalar plus vector)

Gather load first-fault unsigned bytes to vector (vector index)

Gather load with first-faulting behavior of unsigned bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 3 classes: [32-bit unpacked unscaled offset](#) , [32-bit unscaled offset](#) and [64-bit unscaled offset](#)

### 32-bit unpacked unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	xs	0	Zm				0	1	1	Pg			Rn				Zt						
msz<1>msz<0>										U										ff											

**LDFF1B** { **<Zt>.D** }, **<Pg>/Z**, [**<Xn|SP>**, **<Zm>.D**, **<mod>**]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 32-bit unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	0	xs	0	Zm				0	1	1	Pg			Rn				Zt						
										U										ff											

**LDFF1B** { **<Zt>.S** }, **<Pg>/Z**, [**<Xn|SP>**, **<Zm>.S**, **<mod>**]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 64-bit unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	1	0	Zm				1	1	1	Pg			Rn				Zt						
msz<1>msz<0>										U										ff											

**LDFF1B** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
constant integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
- <mod> Is the index extend and shift specifier, encoded in "xs":

xs	<mod>
0	UXTW
1	SXTW

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

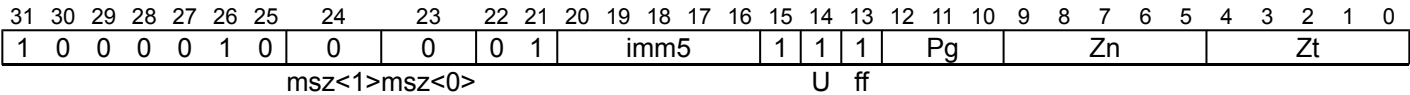
LDFF1B (vector plus immediate)

Gather load first-fault unsigned bytes to vector (immediate index)

Gather load with first-faulting behavior of unsigned bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector. This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

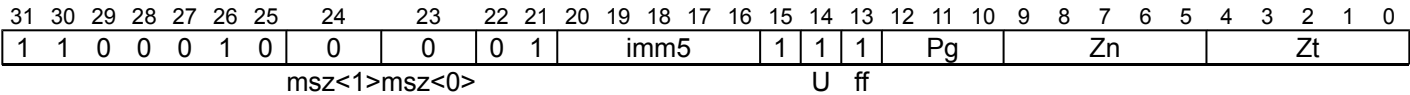
32-bit element



```
LDFF1B { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

64-bit element



```
LDFF1B { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable\_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable\_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

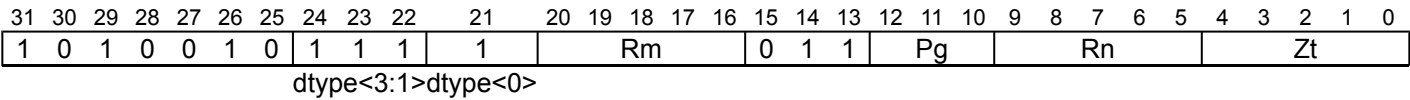
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDFF1D (scalar plus scalar)

Contiguous load first-fault doublewords to vector (scalar index)

Contiguous load with first-faulting behavior of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.



```
LDFF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #3}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.



## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDFF1D (scalar plus vector)

Gather load first-fault doublewords to vector (vector index)

Gather load with first-faulting behavior of doublewords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 4 classes: [32-bit unpacked scaled offset](#) , [32-bit unpacked unscaled offset](#) , [64-bit scaled offset](#) and [64-bit unscaled offset](#)

### 32-bit unpacked scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	1	1	xs	1					Zm	0	1	1		Pg										Zt	
																U ff															

**LDFF1D** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #3]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 3;
```

### 32-bit unpacked unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	1	1	xs	0	Zm				0	1	1	Pg				Rn				Zt					
msz<1>msz<0>																U ff															

**LDFF1D** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

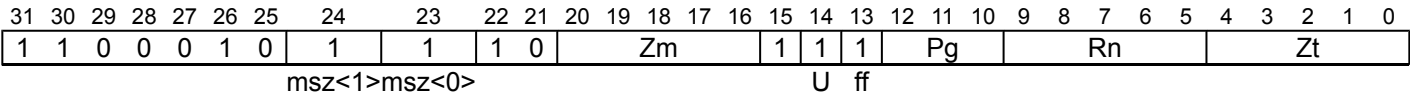
### 64-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	1	1	1	1	Zm				1	1	1	Pg			Rn				Zt						
																U ff															

LDFF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #3]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
constant integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 3;
```

64-bit unscaled offset



LDFF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
constant integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
- <mod> Is the index extend and shift specifier, encoded in "xs":

xs	<mod>
0	UXTW
1	SXTW

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

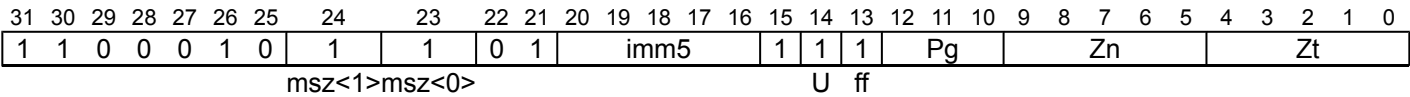
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDFF1D (vector plus immediate)

Gather load first-fault doublewords to vector (immediate index)

Gather load with first-faulting behavior of doublewords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector. This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.



```
LDFF1D { <Zt>.D }, <Pg>/z, [<Zn>.D{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDFF1H (scalar plus scalar)

Contiguous load first-fault unsigned halfwords to vector (scalar index)

Contiguous load with first-faulting behavior of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 3 classes: [16-bit element](#) , [32-bit element](#) and [64-bit element](#)

### 16-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	1	0	1	Rm				0	1	1	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

LDFF1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 16;
boolean unsigned = TRUE;
```

### 32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	1	1	0	Rm				0	1	1	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

LDFF1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
```

### 64-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	1	1	1	Rm				0	1	1	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

```
LDFF1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
```

## Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.



## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDFF1H (scalar plus vector)

Gather load first-fault unsigned halfwords to vector (vector index)

Gather load with first-faulting behavior of unsigned halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 6 classes: [32-bit scaled offset](#) , [32-bit unpacked scaled offset](#) , [32-bit unpacked unscaled offset](#) , [32-bit unscaled offset](#) , [64-bit scaled offset](#) and [64-bit unscaled offset](#)

### 32-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	1	xs	1			Zm			0	1	1		Pg				Rn					Zt		
																U ff															

**LDFF1H** { **<Zt>.S** }, **<Pg>/Z**, [**<Xn|SP>**, **<Zm>.S**, **<mod> #1**]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

### 32-bit unpacked scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	1	xs	1			Zm			0	1	1		Pg				Rn					Zt		
																U ff															

**LDFF1H** { **<Zt>.D** }, **<Pg>/Z**, [**<Xn|SP>**, **<Zm>.D**, **<mod> #1**]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

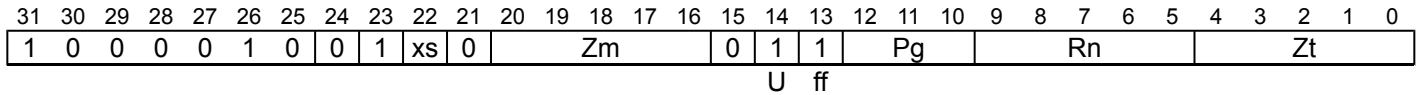
### 32-bit unpacked unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	1	xs	0			Zm			0	1	1		Pg				Rn					Zt		
msz<1>msz<0>																U ff															

**LDFF1H** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

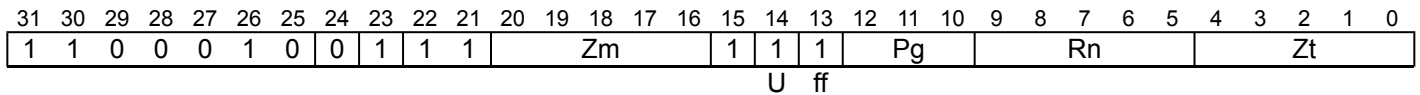
### 32-bit unscaled offset



**LDFF1H** { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

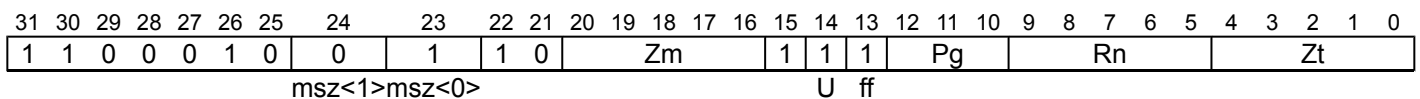
### 64-bit scaled offset



**LDFF1H** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 1;
```

### 64-bit unscaled offset



**LDFF1H** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

**Assembler Symbols**

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
- <mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

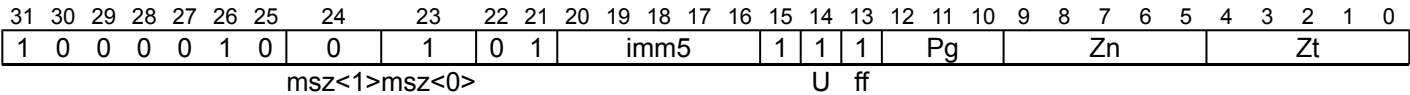
LDF1H (vector plus immediate)

Gather load first-fault unsigned halfwords to vector (immediate index)

Gather load with first-faulting behavior of unsigned halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector. This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: 32-bit element and 64-bit element

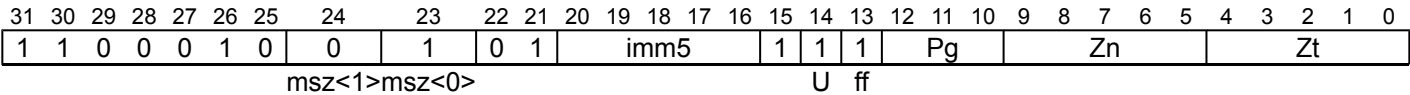
32-bit element



LDF1H { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

64-bit element



LDF1H { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable\_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable\_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDFF1SB (scalar plus scalar)

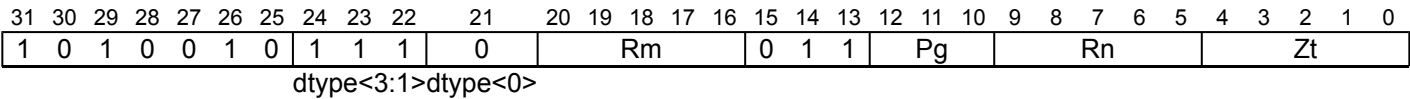
Contiguous load first-fault signed bytes to vector (scalar index)

Contiguous load with first-faulting behavior of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 3 classes: [16-bit element](#) , [32-bit element](#) and [64-bit element](#)

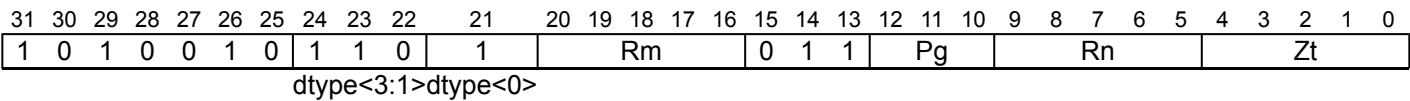
### 16-bit element



LDFF1SB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, <Xm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = FALSE;
```

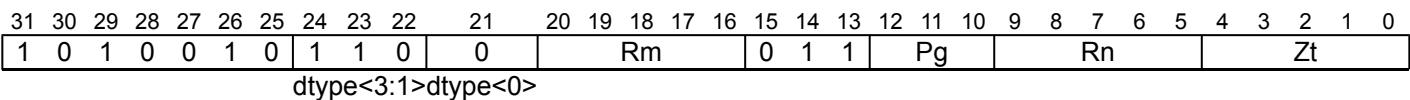
### 32-bit element



LDFF1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
```

### 64-bit element





**LDF1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>}]**

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
```

## Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDFF1SB (scalar plus vector)

Gather load first-fault signed bytes to vector (vector index)

Gather load with first-faulting behavior of signed bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 3 classes: [32-bit unpacked unscaled offset](#) , [32-bit unscaled offset](#) and [64-bit unscaled offset](#)

### 32-bit unpacked unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	xs	0	Zm				0	0	1	Pg			Rn				Zt						
msz<1>msz<0>										U ff																					

**LDFF1SB** { **<Zt>.D** }, **<Pg>/Z**, [**<Xn|SP>**, **<Zm>.D**, **<mod>**]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
constant integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 32-bit unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	0	xs	0	Zm				0	0	1	Pg			Rn				Zt						
										U										ff											

**LDFF1SB** { **<Zt>.S** }, **<Pg>/Z**, [**<Xn|SP>**, **<Zm>.S**, **<mod>**]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
constant integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 64-bit unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	1	0	Zm				1	0	1	Pg			Rn				Zt						
msz<1>msz<0>										U ff																					

**LDFF1SB** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
constant integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
- <mod> Is the index extend and shift specifier, encoded in "xs":

xs	<mod>
0	UXTW
1	SXTW

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDF1SB (vector plus immediate)

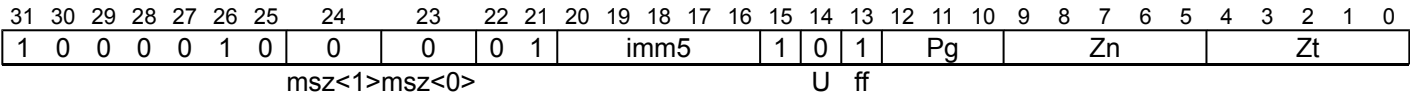
Gather load first-fault signed bytes to vector (immediate index)

Gather load with first-faulting behavior of signed bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

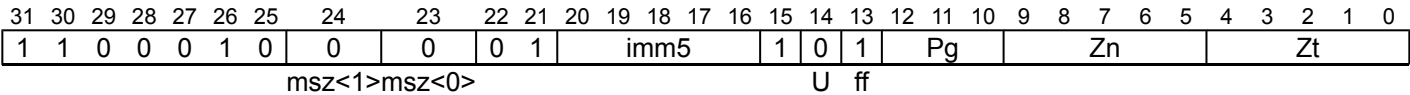
## 32-bit element



LDF1SB { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

## 64-bit element



LDF1SB { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

## Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

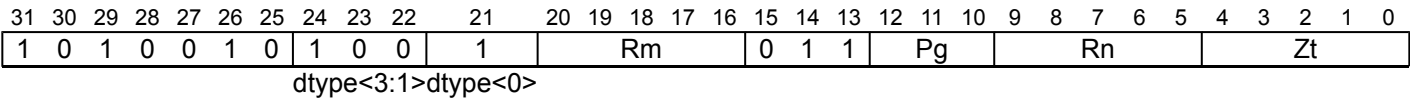
# LDDFF1SH (scalar plus scalar)

Contiguous load first-fault signed halfwords to vector (scalar index)

Contiguous load with first-faulting behavior of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector. This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

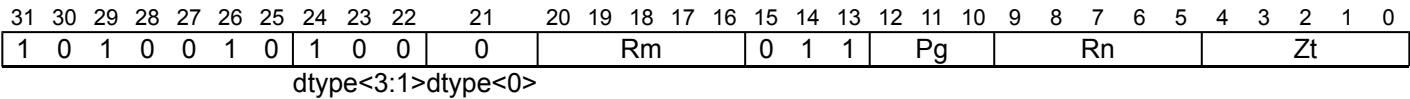
## 32-bit element



LDDFF1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
```

## 64-bit element



LDDFF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
```

## Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.



## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDFF1SH (scalar plus vector)

Gather load first-fault signed halfwords to vector (vector index)

Gather load with first-faulting behavior of signed halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 6 classes: [32-bit scaled offset](#) , [32-bit unpacked scaled offset](#) , [32-bit unpacked unscaled offset](#) , [32-bit unscaled offset](#) , [64-bit scaled offset](#) and [64-bit unscaled offset](#)

### 32-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	1	xs	1			Zm			0	0	1		Pg				Rn					Zt		
																U				ff											

**LDFF1SH** { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
constant integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

### 32-bit unpacked scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	1	xs	1			Zm			0	0	1		Pg				Rn					Zt		
																U				ff											

**LDFF1SH** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

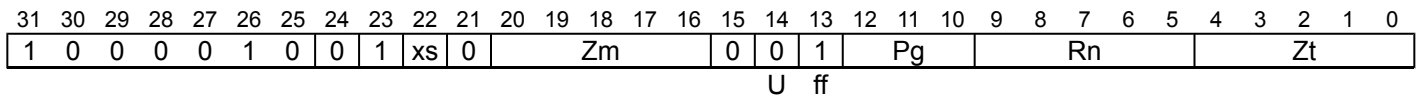
### 32-bit unpacked unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	1	xs	0			Zm			0	0	1		Pg				Rn					Zt		
msz<1>msz<0>																U				ff											

**LDFF1SH** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

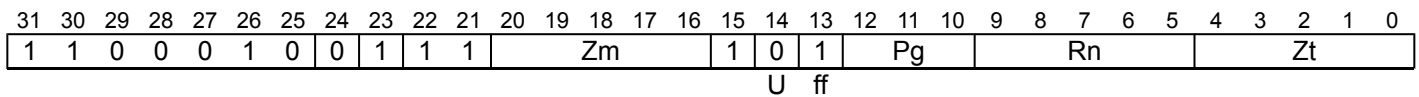
### 32-bit unscaled offset



**LDFF1SH** { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
constant integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

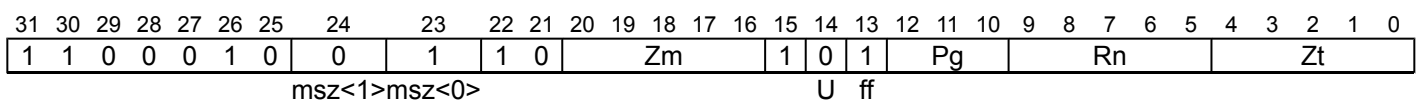
### 64-bit scaled offset



**LDFF1SH** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 1;
```

### 64-bit unscaled offset



LDFF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
- <mod> Is the index extend and shift specifier, encoded in "xs":

xs	<mod>
0	UXTW
1	SXTW

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
        else
            (data, fault) = (Zeros(msize), FALSE);

        // FFR elements set to FALSE following a suppressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR[e, esize] = '0';

        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR[e, esize] == '0';
        if unknown then
            if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
                Elem[result, e, esize] = Extend(data, esize, unsigned);
            elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
                Elem[result, e, esize] = Zeros(esize);
            else // merge
                Elem[result, e, esize] = Elem[orig, e, esize];
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

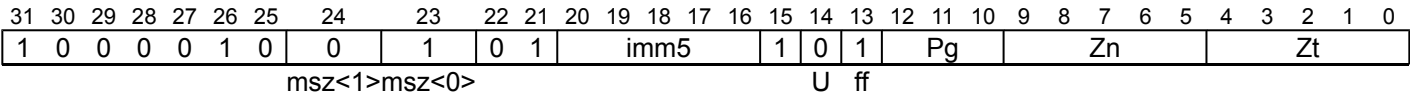
LDFF1SH (vector plus immediate)

Gather load first-fault signed halfwords to vector (immediate index)

Gather load with first-faulting behavior of signed halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector. This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

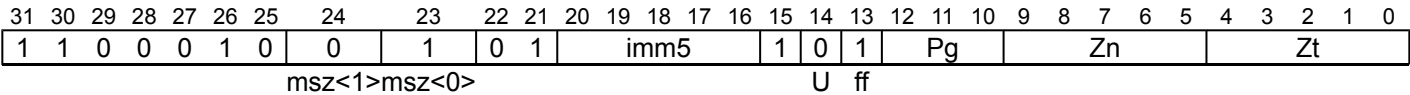
32-bit element



```
LDFF1SH { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

64-bit element



```
LDFF1SH { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
        else
            (data, fault) = (Zeros(msize), FALSE);

        // FFR elements set to FALSE following a suppressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR[e, esize] = '0';

        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR[e, esize] == '0';
        if unknown then
            if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
                Elem[result, e, esize] = Extend(data, esize, unsigned);
            elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
                Elem[result, e, esize] = Zeros(esize);
            else // merge
                Elem[result, e, esize] = Elem[orig, e, esize];
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

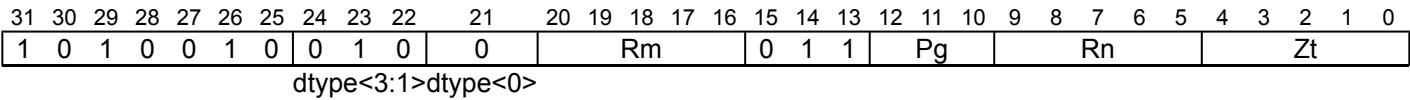
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

**LDFF1SW (scalar plus scalar)**

Contiguous load first-fault signed words to vector (scalar index)

Contiguous load with first-faulting behavior of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.



```
LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
```

**Assembler Symbols**

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.



## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDF1SW (scalar plus vector)

Gather load first-fault signed words to vector (vector index)

Gather load with first-faulting behavior of signed words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 4 classes: [32-bit unpacked scaled offset](#) , [32-bit unpacked unscaled offset](#) , [64-bit scaled offset](#) and [64-bit unscaled offset](#)

### 32-bit unpacked scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	1	0	xs	1	Zm					0	0	1	Pg				Rn				Zt				
																U ff															

**LDF1SW** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

### 32-bit unpacked unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	1	0	xs	0	Zm			0	0	1	Pg			Rn			Zt								
msz<1>msz<0>																U ff															

**LDF1SW** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

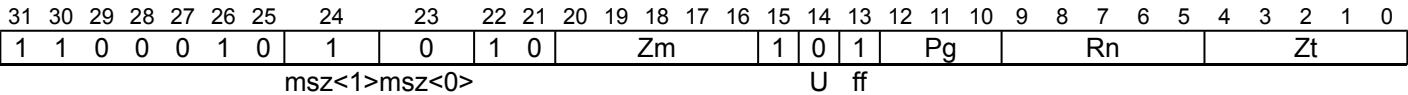
### 64-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	1	0	1	1	Zm			1	0	1	Pg			Rn			Zt								
																U ff															

LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 2;
```

64-bit unscaled offset



LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
- <mod> Is the index extend and shift specifier, encoded in "xs":

xs	<mod>
0	UXTW
1	SXTW

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable\_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable\_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

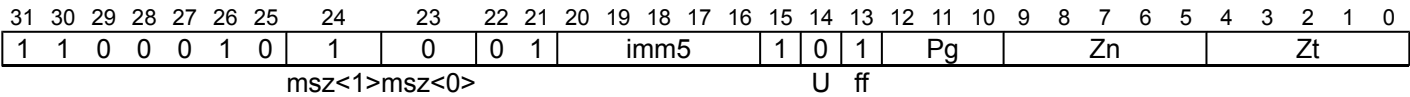
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDFF1SW (vector plus immediate)

Gather load first-fault signed words to vector (immediate index)

Gather load with first-faulting behavior of signed words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector. This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.



```
LDFF1SW { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable\_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable\_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

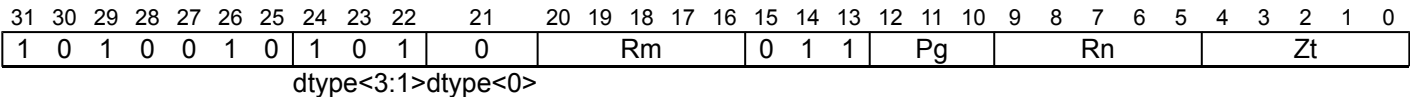
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDF1W (scalar plus scalar)

Contiguous load first-fault unsigned words to vector (scalar index)

Contiguous load with first-faulting behavior of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector. This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled. It has encodings from 2 classes: 32-bit element and 64-bit element

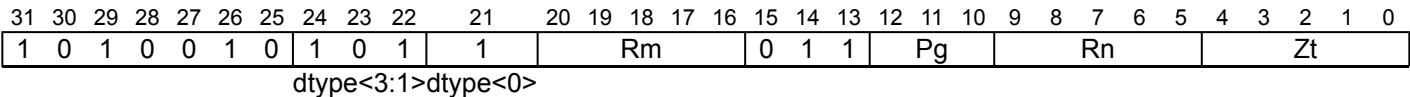
32-bit element



```
LDF1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
```

64-bit element



```
LDF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## LDFF1W (scalar plus vector)

Gather load first-fault unsigned words to vector (vector index)

Gather load with first-faulting behavior of unsigned words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 6 classes: [32-bit scaled offset](#) , [32-bit unpacked scaled offset](#) , [32-bit unpacked unscaled offset](#) , [32-bit unscaled offset](#) , [64-bit scaled offset](#) and [64-bit unscaled offset](#)

### 32-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1	0	xs	1			Zm			0	1	1		Pg				Rn					Zt		
																U ff															

**LDFF1W** { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

### 32-bit unpacked scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	1	0	xs	1			Zm			0	1	1		Pg				Rn					Zt		
																U ff															

**LDFF1W** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

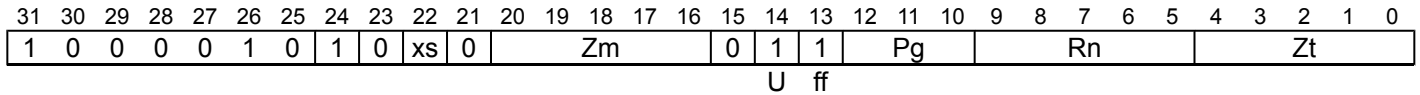
### 32-bit unpacked unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	1	0	xs	0			Zm			0	1	1		Pg				Rn					Zt		
msz<1>msz<0>																U ff															

**LDFF1W** { **<Zt>.D** }, **<Pg>/Z**, [**<Xn|SP>**, **<Zm>.D**, **<mod>**]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

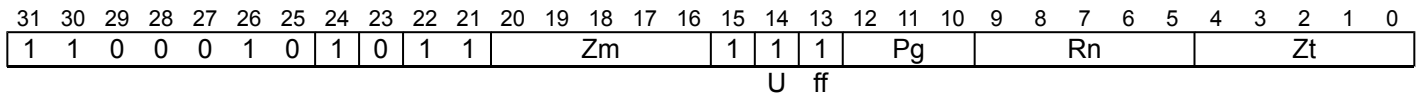
### 32-bit unscaled offset



**LDFF1W** { **<Zt>.S** }, **<Pg>/Z**, [**<Xn|SP>**, **<Zm>.S**, **<mod>**]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
constant integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

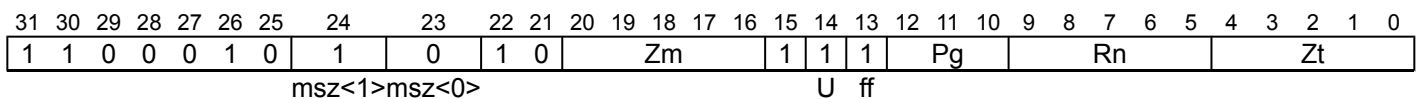
### 64-bit scaled offset



**LDFF1W** { **<Zt>.D** }, **<Pg>/Z**, [**<Xn|SP>**, **<Zm>.D**, **LSL #2**]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 2;
```

### 64-bit unscaled offset



LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
- <mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable\_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable\_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

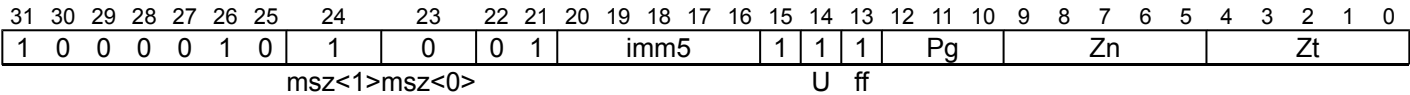
LDDFF1W (vector plus immediate)

Gather load first-fault unsigned words to vector (immediate index)

Gather load with first-faulting behavior of unsigned words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector. This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

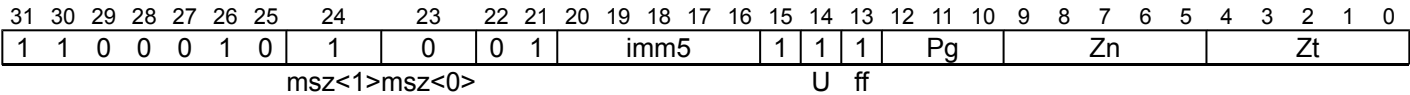
32-bit element



```
LDDFF1W { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

64-bit element



```
LDDFF1W { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVEFF(contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

assert accdesc.first;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if accdesc.first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, accdesc];
            accdesc.first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable\_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable\_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNF1B

Contiguous load non-fault unsigned bytes to vector (immediate index)

Contiguous load with non-faulting behavior of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 4 classes: [8-bit element](#) , [16-bit element](#) , [32-bit element](#) and [64-bit element](#)

### 8-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	0	0	0	1	imm4				1	0	1	Pg			Rn				Zt					
dtype<3:1>dtype<0>																															

LDNF1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

### 16-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	0	0	1	1	imm4			1	0	1	Pg			Rn				Zt						
dtype<3:1>dtype<0>																															

LDNF1B { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

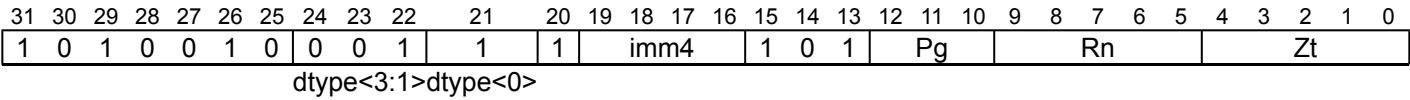
### 32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	0	1	0	1		imm4			1	0	1	Pg			Rn				Zt					
dtype<3:1>dtype<0>																															

LDNFB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

64-bit element



LDNFB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.



## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVENF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

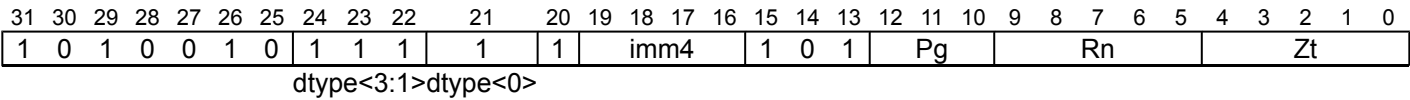
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDNF1D

Contiguous load non-fault doublewords to vector (immediate index)

Contiguous load with non-faulting behavior of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.



```
LDNF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVENF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNF1H

Contiguous load non-fault unsigned halfwords to vector (immediate index)

Contiguous load with non-faulting behavior of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 3 classes: [16-bit element](#) , [32-bit element](#) and [64-bit element](#)

### 16-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	1	0	1	1	imm4				1	0	1	Pg			Rn				Zt					
dtype<3:1>dtype<0>																															

LDNF1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

### 32-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	1	1	0	1	imm4				1	0	1	Pg			Rn				Zt					
dtype<3:1>dtype<0>																															

LDNF1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

### 64-bit element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	1	1	1	1	imm4				1	0	1	Pg			Rn				Zt					
dtype<3:1>dtype<0>																															

```
LDNF1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

## Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVENF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDNF1SB

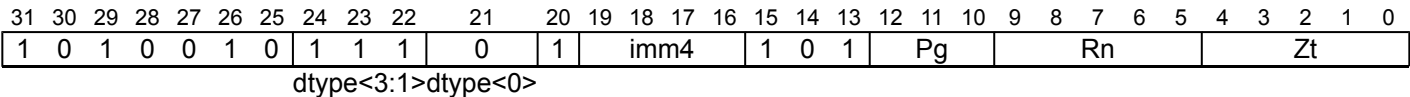
Contiguous load non-fault signed bytes to vector (immediate index)

Contiguous load with non-faulting behavior of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 3 classes: [16-bit element](#) , [32-bit element](#) and [64-bit element](#)

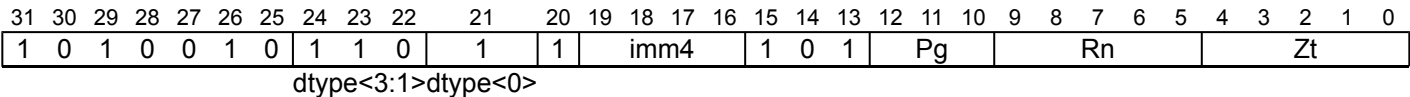
## 16-bit element



LDNF1SB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

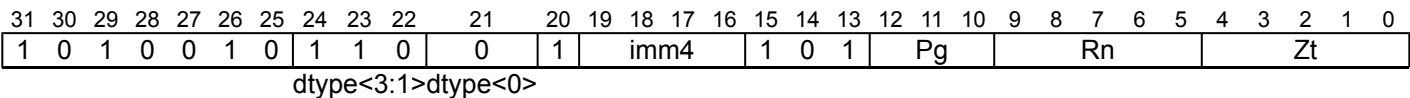
## 32-bit element



LDNF1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

## 64-bit element



```
LDNF1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

## Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.



## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVENF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

// FFR elements set to FALSE following a suppressed access/fault
faulted = faulted || fault;
if faulted then
    ElemFFR[e, esize] = '0';

// Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
unknown = unknown || ElemFFR[e, esize] == '0';
if unknown then
    if !fault && ConstrainUnpredictableBool(Unpredictable\_SVELDNFDATA) then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable\_SVELDNFZERO) then
        Elem[result, e, esize] = Zeros(esize);
    else // merge
        Elem[result, e, esize] = Elem[orig, e, esize];
else
    Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDNF1SH

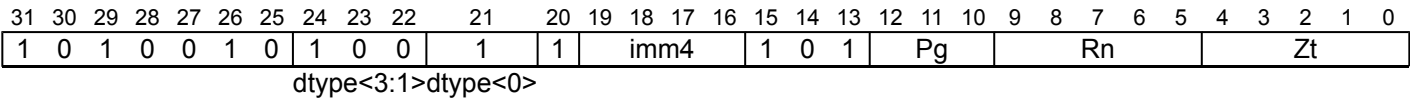
Contiguous load non-fault signed halfwords to vector (immediate index)

Contiguous load with non-faulting behavior of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

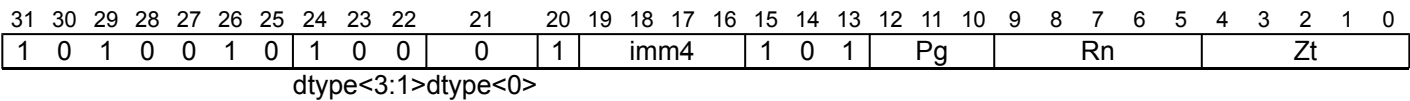
## 32-bit element



LDNF1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

## 64-bit element



LDNF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

## Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVENF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

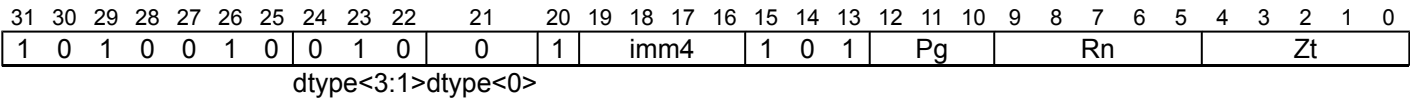
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDNF1SW

Contiguous load non-fault signed words to vector (immediate index)

Contiguous load with non-faulting behavior of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.



```
LDNF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVENF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDNF1W

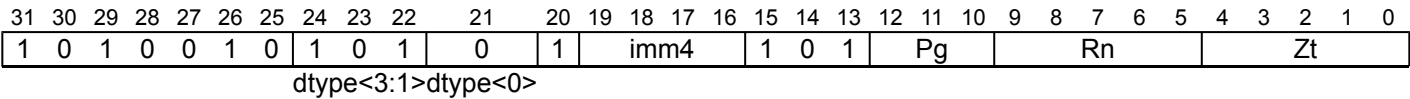
Contiguous load non-fault unsigned words to vector (immediate index)

Contiguous load with non-faulting behavior of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

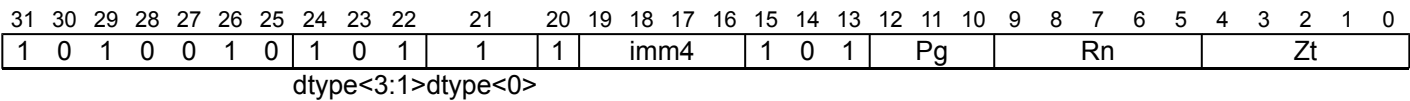
32-bit element



LDNF1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

64-bit element



LDNF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
boolean contiguous = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVENF(contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, accdesc];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1B (scalar plus immediate, consecutive registers)

Contiguous load non-temporal of bytes to multiple consecutive vectors (immediate index)

Contiguous load non-temporal of bytes to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	1	0	0	imm4				0	0	0	PNg				Rn				Zt				1
																	msz<1>msz<0>													N	

LDNT1B { <Zt1>.B-<Zt2>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 8;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	1	0	0	0	0	0	0	1	0	0	imm4				1	0	0	PNg				Rn				Zt				0	1
																	msz<1>msz<0>													N		

LDNT1B { <Zt1>.B-<Zt4>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 8;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.  
For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.



<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1B (scalar plus immediate, single register)

Contiguous load non-temporal bytes to vector (immediate index)

Contiguous load non-temporal of bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	0	0	0	0	imm4			1	1	1	Pg			Rn			Zt							
msz<1>msz<0>																															

```
LDNT1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1B (scalar plus scalar, consecutive registers)

Contiguous load non-temporal of bytes to multiple consecutive vectors (scalar index)

Contiguous load non-temporal of bytes to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm			0	0	0	PNg	Rn			Zt			1							
																msz<1>msz<0>										N					

LDNT1B { <Zt1>.B-<Zt2>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 8;
```

### Four registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm			1	0	0	PNg	Rn			Zt			0	1						
																msz<1>msz<0>										N					

LDNT1B { <Zt1>.B-<Zt4>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 8;
```

### Assembler Symbols

<Zt1>	For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.  For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1B (scalar plus scalar, single register)

Contiguous load non-temporal bytes to vector (scalar index)

Contiguous load non-temporal of bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	0	0	0			Rm			1	1	0		Pg				Rn					Zt		
msz<1>msz<0>																															

**LDNT1B** { **<Zt>.B** }, **<Pg>/Z**, [**<Xn|SP>**, **<Xm>**]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
```

### Assembler Symbols

- <Zt>** Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm>** Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1B (vector plus scalar)

Gather load non-temporal unsigned bytes

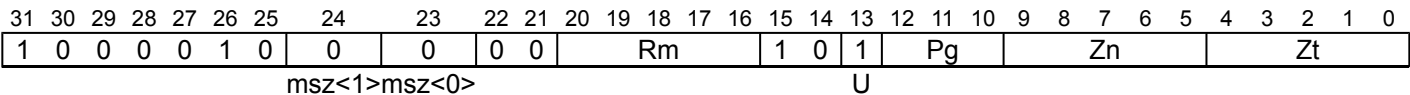
Gather load non-temporal of unsigned bytes to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit unscaled offset](#) and [64-bit unscaled offset](#)

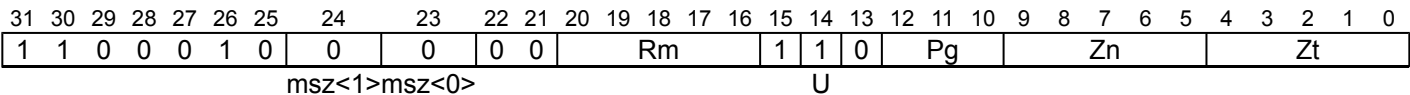
### 32-bit unscaled offset



LDNT1B { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
```

### 64-bit unscaled offset



LDNT1B { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.



## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1D (scalar plus immediate, consecutive registers)

Contiguous load non-temporal of doublewords to multiple consecutive vectors (immediate index)

Contiguous load non-temporal of doublewords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	1	0	0	imm4				0	1	1	PNg				Rn				Zt			1	
																	msz<1>msz<0>													N	

LDNT1D { <Zt1>.D-<Zt2>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	1	0	0	imm4				1	1	1	PNg				Rn				Zt			0	1
																	msz<1>msz<0>													N	

LDNT1D { <Zt1>.D-<Zt4>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.
- For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1D (scalar plus immediate, single register)

Contiguous load non-temporal doublewords to vector (immediate index)

Contiguous load non-temporal of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	1	1	0	0	0	imm4				1	1	1	Pg			Rn				Zt					
msz<1>msz<0>																															

```
LDNT1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1D (scalar plus scalar, consecutive registers)

Contiguous load non-temporal of doublewords to multiple consecutive vectors (scalar index)

Contiguous load non-temporal of doublewords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	0					Rm	0	1	1	PNg									Zt		1	
																	msz<1>msz<0>										N				

LDNT1D { <Zt1>.D-<Zt2>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 64;
```

### Four registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	0					Rm	1	1	1	PNg									Zt	0	1	
																	msz<1>msz<0>										N				

LDNT1D { <Zt1>.D-<Zt4>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 64;
```

### Assembler Symbols

<Zt1>	For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.  For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

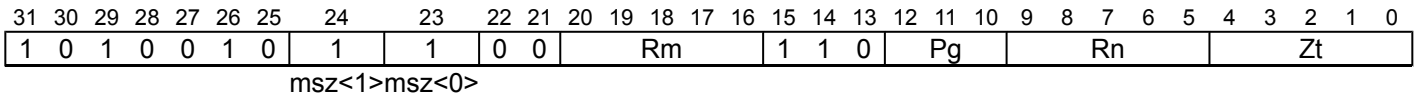
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1D (scalar plus scalar, single register)

Contiguous load non-temporal doublewords to vector (scalar index)

Contiguous load non-temporal of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.



**LDNT1D** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```



## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

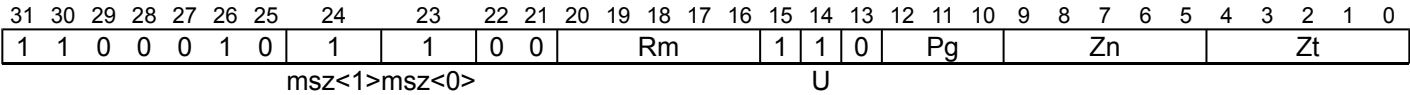
LDNT1D (vector plus scalar)

Gather load non-temporal unsigned doublewords

Gather load non-temporal of doublewords to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.



```
LDNT1D { <Zt>.D }, <Pg>/z, [<Zn>.D{, <Xm>}]
```

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1H (scalar plus immediate, consecutive registers)

Contiguous load non-temporal of halfwords to multiple consecutive vectors (immediate index)

Contiguous load non-temporal of halfwords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	1	0	0	imm4				0	0	1	PNg				Rn				Zt			1	
																msz<1>msz<0>														N	

LDNT1H { <Zt1>.H-<Zt2>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 16;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	1	0	0	imm4				1	0	1	PNg				Rn				Zt			0	1
																msz<1>msz<0>														N	

LDNT1H { <Zt1>.H-<Zt4>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 16;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.
- For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1H (scalar plus immediate, single register)

Contiguous load non-temporal halfwords to vector (immediate index)

Contiguous load non-temporal of halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	1	0	0	0	imm4				1	1	1	Pg			Rn				Zt					
msz<1>msz<0>																															

```
LDNT1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
```

### Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1H (scalar plus scalar, consecutive registers)

Contiguous load non-temporal of halfwords to multiple consecutive vectors (scalar index)

Contiguous load non-temporal of halfwords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm			0	0	1	PNg		Rn			Zt			1						
																msz<1>msz<0>														N	

LDNT1H { <Zt1>.H-<Zt2>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 16;
```

### Four registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm			1	0	1	PNg		Rn			Zt			0	1					
																msz<1>msz<0>														N	

LDNT1H { <Zt1>.H-<Zt4>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 16;
```

### Assembler Symbols

<Zt1>	For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.  For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.



## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1H (scalar plus scalar, single register)

Contiguous load non-temporal halfwords to vector (scalar index)

Contiguous load non-temporal of halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	0	1	0	0	Rm				1	1	0	Pg			Rn				Zt						
msz<1>msz<0>																															

msz<1>msz<0>

**LDNT1H** { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDNT1H (vector plus scalar)

Gather load non-temporal unsigned halfwords

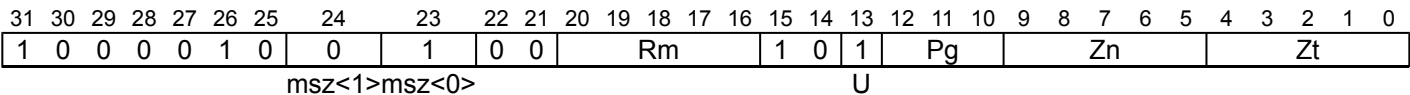
Gather load non-temporal of unsigned halfwords to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit unscaled offset](#) and [64-bit unscaled offset](#)

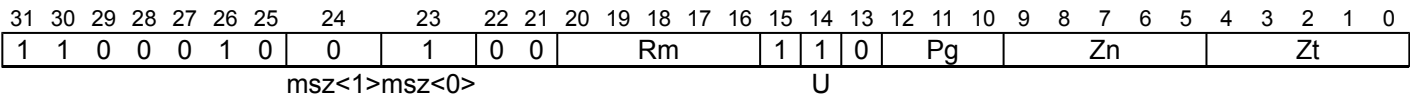
32-bit unscaled offset



```
LDNT1H { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]
```

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
```

64-bit unscaled offset



```
LDNT1H { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]
```

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDNT1SB

Gather load non-temporal signed bytes

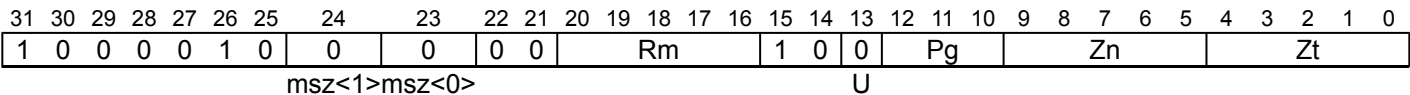
Gather load non-temporal of signed bytes to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit unscaled offset](#) and [64-bit unscaled offset](#)

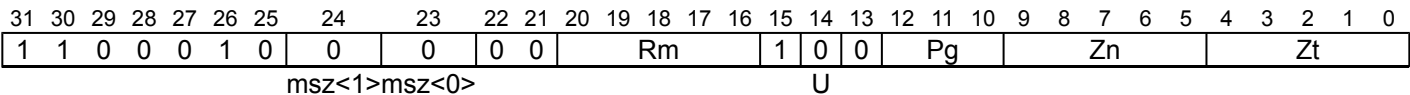
## 32-bit unscaled offset



LDNT1SB { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
```

## 64-bit unscaled offset



LDNT1SB { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
```

## Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDNT1SH

Gather load non-temporal signed halfwords

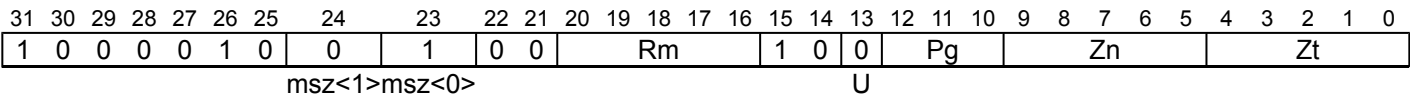
Gather load non-temporal of signed halfwords to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit unscaled offset](#) and [64-bit unscaled offset](#)

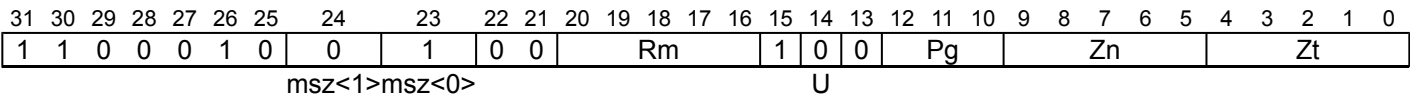
## 32-bit unscaled offset



LDNT1SH { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
```

## 64-bit unscaled offset



LDNT1SH { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
```

## Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.



## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

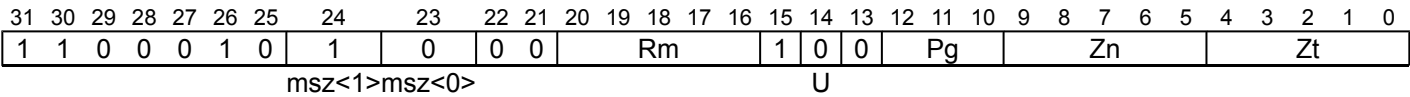
LDNT1SW

Gather load non-temporal signed words

Gather load non-temporal of signed words to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.



```
LDNT1SW { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]
```

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1W (scalar plus immediate, consecutive registers)

Contiguous load non-temporal of words to multiple consecutive vectors (immediate index)

Contiguous load non-temporal of words to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	1	0	0	imm4				0	1	0	PNg				Rn				Zt			1	
																	msz<1>msz<0>												N		

```
LDNT1W { <Zt1>.S-<Zt2>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 32;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	1	0	0	imm4				1	1	0	PNg				Rn				Zt			0	1
																	msz<1>msz<0>										N				

```
LDNT1W { <Zt1>.S-<Zt4>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 32;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.
- For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

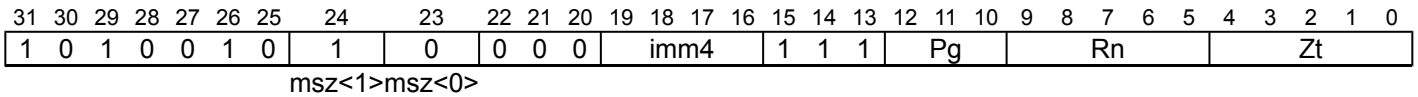
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1W (scalar plus immediate, single register)

Contiguous load non-temporal words to vector (immediate index)

Contiguous load non-temporal of words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.



```
LDNT1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1W (scalar plus scalar, consecutive registers)

Contiguous load non-temporal of words to multiple consecutive vectors (scalar index)

Contiguous load non-temporal of words to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	0					Rm	0	1	0			PNg								Zt		1
																msz<1>msz<0>															N

LDNT1W { <Zt1>.S-<Zt2>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 32;
```

### Four registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	0					Rm	1	1	0			PNg								Zt	0	1
																msz<1>msz<0>															N

LDNT1W { <Zt1>.S-<Zt4>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 32;
```

### Assembler Symbols

<Zt1>	For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.  For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.



## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

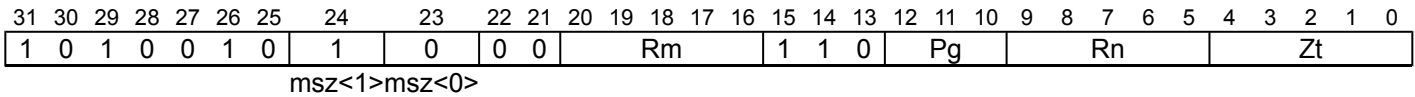
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1W (scalar plus scalar, single register)

Contiguous load non-temporal words to vector (scalar index)

Contiguous load non-temporal of words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.



```
LDNT1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDNT1W (vector plus scalar)

Gather load non-temporal unsigned words

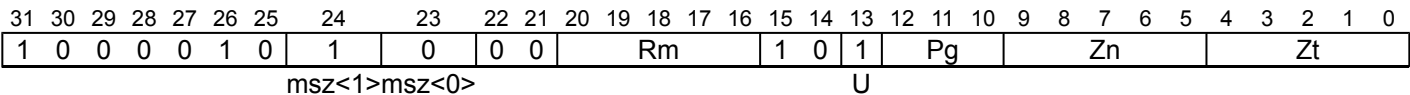
Gather load non-temporal of unsigned words to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit unscaled offset](#) and [64-bit unscaled offset](#)

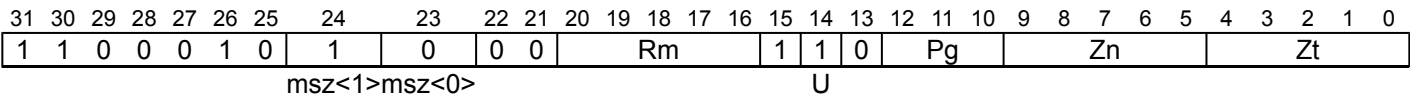
## 32-bit unscaled offset



LDNT1W { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
```

## 64-bit unscaled offset



LDNT1W { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
```

## Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        data = Mem[addr, mbytes, accdesc];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LDR (predicate)

Load predicate register

Load a predicate register from a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current predicate register size in bytes. This instruction is unpredicated. The load is performed as contiguous byte accesses, each containing 8 consecutive predicate bits in ascending element order, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then a general-purpose base register must be aligned to 2 bytes. For programmer convenience, an assembler must also accept a predicate-as-counter register name for the destination predicate register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1	1	0	imm9h						0	0	0	imm9l			Rn			0	Pt					

```
LDR <Pt>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Pt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);
```

## Assembler Symbols

- <Pt> Is the name of the destination scalable predicate register, encoded in the "Pt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = PL DIV 8;
bits(64) base;
integer offset = imm * elements;
bits(PL) result;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if n == 31 then
    CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

boolean aligned = IsAligned(base + offset, 2);

if !aligned && AlignmentEnforced() then
    AArch64.Abort(base + offset, AlignmentFault(accdesc));

for e = 0 to elements-1
    Elem[result, e, 8] = AArch64.MemSingle(base + offset, 1, accdesc, aligned);
    offset = offset + 1;

P[t, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDR (vector)

Load vector register

Load a vector register from a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current vector register size in bytes. This instruction is unpredicated.

The load is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1	1	0	imm9h						0	1	0	imm9l			Rn					Zt				

LDR <Zt>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 8;
bits(64) base;
integer offset = imm * elements;
bits(VL) result;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if n == 31 then
    CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

boolean aligned = IsAligned(base + offset, 16);

if !aligned && AlignmentEnforced() then
    AArch64.Abort(base + offset, AlignmentFault(accdesc));

for e = 0 to elements-1
    Elem[result, e, 8] = AArch64.MemSingle(base + offset, 1, accdesc, aligned);
    offset = offset + 1;

Z[t, VL] = result;
```

### Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

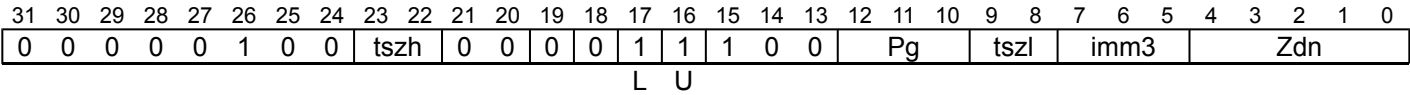




LSL (immediate, predicated)

Logical shift left by immediate (predicated)

Shift left by immediate each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.



```
LSL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer PL = VL DIV 8;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = LSL(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSL (immediate, unpredicated)

Logical shift left by immediate (unpredicated)

Shift left by immediate each element of the source vector, and place the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	tszh	1	tszl	imm3	1	0	0	1	1	1														

LSL <Zd>.<T>, <Zn>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    Elem[result, e, esize] = LSL(element1, shift);

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

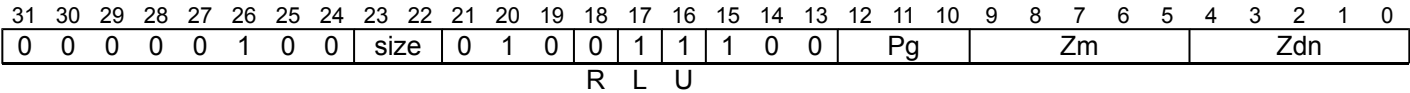
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



LSL (vectors)

Logical shift left by vector (predicated)

Shift left active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.



```
LSL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        integer shift = Min(UInt(element2), esize);
        Elem[result, e, esize] = LSL(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

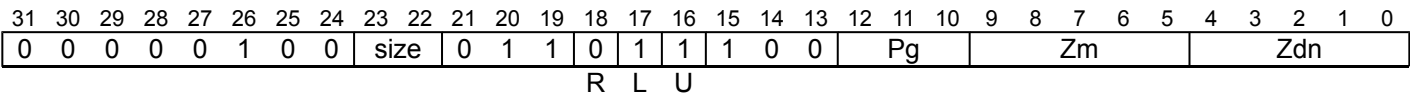
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSL (wide elements, predicated)

Logical shift left by 64-bit wide elements (predicated)

Shift left active elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.



```
LSL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.D
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	RESERVED

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
    integer shift = Min(UInt(element2), esize);
    Elem[result, e, esize] = LSL(element1, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:



- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and destination element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSL (wide elements, unpredicated)

Logical shift left by 64-bit wide elements (unpredicated)

Shift left all elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and place the first in the corresponding elements of the destination vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1				Zm			1	0	0	0	1	1				Zn				Zd		

LSL <Zd>.<T>, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	RESERVED

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
    integer shift = Min(UInt(element2), esize);
    Elem[result, e, esize] = LSL(element1, shift);

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

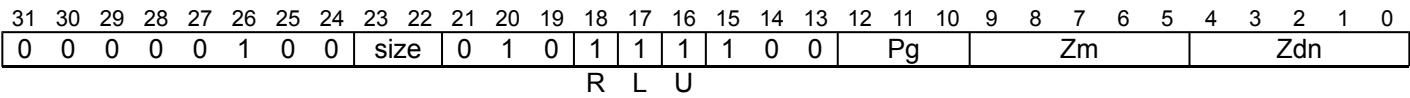
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



LSLR

Reversed logical shift left by vector (predicated)

Reversed shift left active elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.



```
LSLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    integer shift = Min(UInt(element1), esize);
    Elem[result, e, esize] = LSL(element2, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

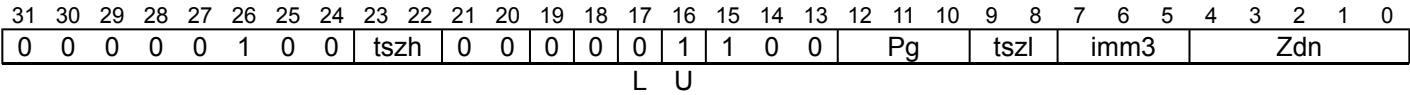
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSR (immediate, predicated)

Logical shift right by immediate (predicated)

Shift right by immediate, inserting zeroes, each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.



LSR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

- <Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- <T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer PL = VL DIV 8;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ActivePredicateElement(mask, e, esize) then
    Elem[result, e, esize] = LSR(element1, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

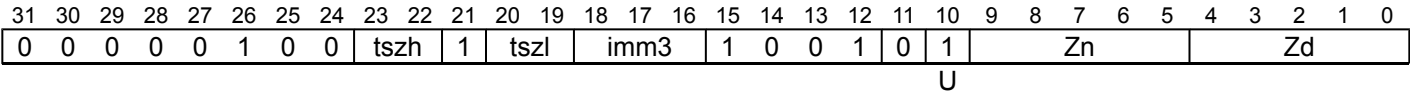
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSR (immediate, unpredicated)

Logical shift right by immediate (unpredicated)

Shift right by immediate, inserting zeroes, each element of the source vector, and place the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



```
LSR <Zd>.<T>, <Zn>.<T>, #<const>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    Elem[result, e, esize] = LSR(element1, shift);

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

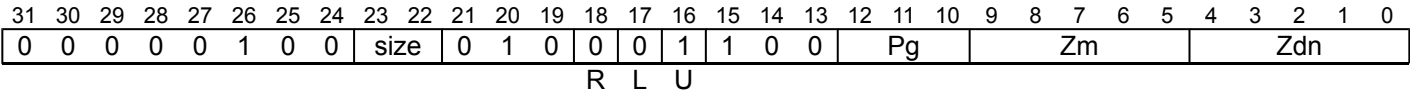




LSR (vectors)

Logical shift right by vector (predicated)

Shift right, inserting zeroes, active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.



```
LSR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        integer shift = Min(UInt(element2), esize);
        Elem[result, e, esize] = LSR(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSR (wide elements, predicated)

Logical shift right by 64-bit wide elements (predicated)

Shift right, inserting zeroes, active elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0	0	0	0	0	1	0	0	size	0	1	1	0	0	1	1	0	0	Pg			Zm				Zdn										
R																L		U																	

LSR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	RESERVED

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
        integer shift = Min(UInt(element2), esize);
        Elem[result, e, esize] = LSR(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and destination element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

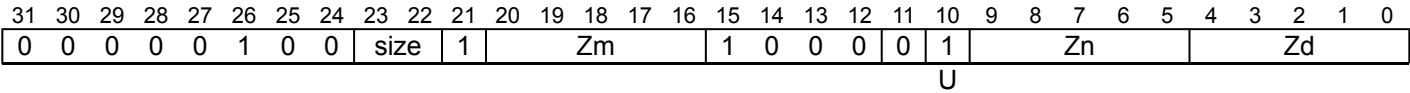
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSR (wide elements, unpredicated)

Logical shift right by 64-bit wide elements (unpredicated)

Shift right, inserting zeroes, all elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and place the first in the corresponding elements of the destination vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. This instruction is unpredicated.



```
LSR <Zd>.<T>, <Zn>.<T>, <Zm>.<D>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	RESERVED

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
    integer shift = Min(UInt(element2), esize);
    Elem[result, e, esize] = LSR(element1, shift);

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

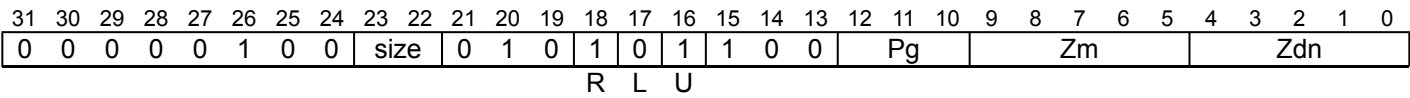
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSRR

Reversed logical shift right by vector (predicated)

Reversed shift right, inserting zeroes, active elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.



```
LSRR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    integer shift = Min(UInt(element1), esize);
    Elem[result, e, esize] = LSR(element2, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:



- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

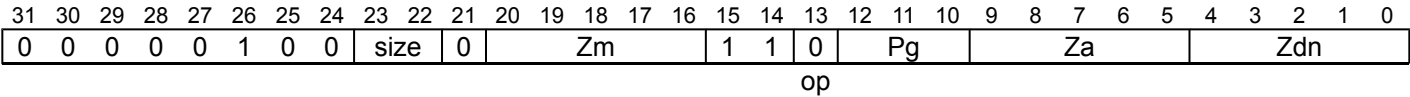
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MAD

Multiply-add vectors (predicated), writing multiplicand [Zdn = Za + Zdn \* Zm]

Multiply the corresponding active elements of the first and second source vectors and add to elements of the third (addend) vector. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.



```
MAD <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean sub_op = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Za> Is the name of the third source scalable vector register, encoded in the "Za" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer product = element1 * element2;
    if sub_op then
      Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MATCH

Detect any matching elements, setting the condition flags

This instruction compares each active 8-bit or 16-bit character in the first source vector with all of the characters in the corresponding 128-bit segment of the second source vector. Where the first source element detects any matching characters in the second segment it places true in the corresponding element of the destination predicate, otherwise false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size	1	Zm						1	0	0	Pg			Zn				0	Pd				

MATCH <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() then UNDEFINED;
if size IN {'lx'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Pd);
integer n = UInt(Zn);
integer m = UInt(Zm);
```

Assembler Symbols

- <Pd>

Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <T>

Is the size specifier, encoded in "size<0>":

size<0>	<T>
0	B
1	H
- <Pg>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn>

Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm>

Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer segmentbase = e - (e MOD eltspersegment);
        Elem[result, e, psize] = ZeroExtend('0', psize);
        bits(esize) element1 = Elem[operand1, e, esize];
        for i = segmentbase to (segmentbase + eltspersegment) - 1
            bits(esize) element2 = Elem[operand2, i, esize];
            if element1 == element2 then
                Elem[result, e, psize] = ZeroExtend('1', psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## MLA (indexed)

Multiply-add to accumulator (indexed)

Multiply all integer elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively added to the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

It has encodings from 3 classes: [16-bit](#) , [32-bit](#) and [64-bit](#)

### 16-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	0	1	0	0	0	i3h	1		i3l		Zm		0	0	0	0	1	0		Zn					Zda				
																					S											

**MLA** <Zda>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	0	1	i2	Zm			0	0	0	0	1	0	Zn				Zda						
size<1>size<0>											S																				

**MLA** <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### 64-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	1	1	i1	Zm			0	0	0	0	1	0	Zn				Zda						
size<1>size<0>											S																				

**MLA** <Zda>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

## Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field. For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

## Operation

```

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, s, esize]);
    bits(esize) product = (element1 * element2) < esize-1:0 >;
    Elem[result, e, esize] = Elem[result, e, esize] + product;

Z[da, VL] = result;

```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

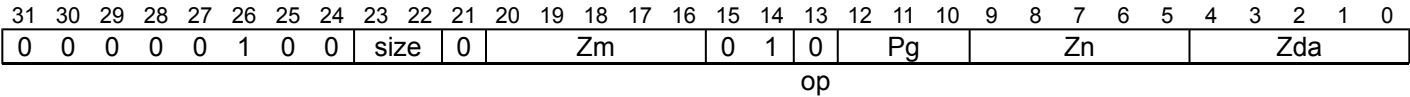
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MLA (vectors)

Multiply-add vectors (predicated), writing addend [Zda = Zda + Zn \* Zm]

Multiply the corresponding active elements of the first and second source vectors and add to elements of the third source (addend) vector. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.



MLA <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean sub_op = FALSE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer product = element1 * element2;
    if sub_op then
      Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
  else
    Elem[result, e, esize] = Elem[operand3, e, esize];

Z[da, VL] = result;
```



## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## MLS (indexed)

Multiply-subtract from accumulator (indexed)

Multiply all integer elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively subtracted from the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

It has encodings from 3 classes: [16-bit](#) , [32-bit](#) and [64-bit](#)

### 16-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	0	1	0	0	0	i3h	1	i3l	Zm			0	0	0	0	1	1	Zn				Zda							
																					S											

**MLS** <Zda>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	0	1	i2	Zm			0	0	0	0	1	1	Zn				Zda						
size<1>size<0>											S																				

**MLS** <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### 64-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	1	1	i1	Zm			0	0	0	0	1	1	Zn				Zda						
size<1>size<0>											S																				

**MLS** <Zda>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

## Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field. For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

## Operation

```

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, s, esize]);
    bits(esize) product = (element1 * element2) < esize-1:0 >;
    Elem[result, e, esize] = Elem[result, e, esize] - product;

Z[da, VL] = result;

```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

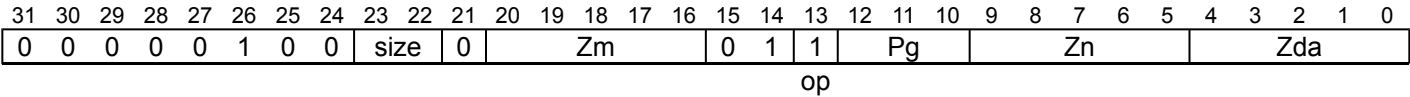
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MLS (vectors)

Multiply-subtract vectors (predicated), writing addend [Zda = Zda - Zn \* Zm]

Multiply the corresponding active elements of the first and second source vectors and subtract from elements of the third source (addend) vector. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.



```
MLS <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean sub_op = TRUE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer product = element1 * element2;
    if sub_op then
      Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
  else
    Elem[result, e, esize] = Elem[operand3, e, esize];

Z[da, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV

Move logical bitmask immediate to vector (unpredicated)

Unconditionally broadcast the logical bitmask immediate into each element of the destination vector. This instruction is unpredicated. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits.

This is an alias of [DUPM](#). This means:

- The encodings in this description are named to match the encodings of [DUPM](#).
- The description of [DUPM](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	1	1	0	0	0	0	imm13												Zd					

MOV <Zd>.<T>, #<const>

is equivalent to

[DUPM](#) <Zd>.<T>, #<const>

and is the preferred disassembly when `SVEMoveMaskPreferred(imm13)`.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "imm13<12>:imm13<5:0>":

imm13<12>	imm13<5:0>	<T>
0	0xxxxx	S
0	10xxxx	H
0	110xxx	B
0	1110xx	B
0	11110x	B
0	111110	RESERVED
0	111111	RESERVED
1	xxxxxx	D

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

Operation

The description of [DUPM](#) gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# MOV

Move predicate (unpredicated)

Read all elements from the source predicate and place in the destination predicate. This instruction is unpredicated. Does not set the condition flags.  
For programmer convenience, an assembler must also accept predicate-as-counter register names for the source and destination predicate registers.

This is an alias of [ORR \(predicates\)](#). This means:

- The encodings in this description are named to match the encodings of [ORR \(predicates\)](#).
- The description of [ORR \(predicates\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	1	0	0	0	Pm			0	1	Pg			0	Pn			0	Pd			S			

MOV <Pd>.B, <Pn>.B

is equivalent to

ORR <Pd>.B, <Pn>/Z, <Pn>.B, <Pn>.B

and is the preferred disassembly when `S == '0' && Pn == Pm && Pm == Pg`.

## Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

## Operation

The description of [ORR \(predicates\)](#) gives the operational pseudocode for this instruction.

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (immediate, predicated, merging)

Move signed integer immediate to vector elements (merging)

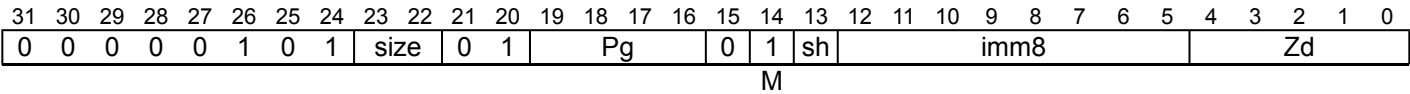
Move a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<simm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This is an alias of [CPY \(immediate, merging\)](#). This means:

- The encodings in this description are named to match the encodings of [CPY \(immediate, merging\)](#).
- The description of [CPY \(immediate, merging\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



MOV <Zd>.<T>, <Pg>/M, #<imm>{, <shift>}

is equivalent to

CPY <Zd>.<T>, <Pg>/M, #<imm>{, <shift>}

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<imm> Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #8

Operation

The description of [CPY \(immediate, merging\)](#) gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:



- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# MOV (immediate, predicated, zeroing)

Move signed integer immediate to vector elements (zeroing)

Move a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register are set to zero.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This is an alias of [CPY \(immediate, zeroing\)](#). This means:

- The encodings in this description are named to match the encodings of [CPY \(immediate, zeroing\)](#).
- The description of [CPY \(immediate, zeroing\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size		0	1	Pg				0	0	sh	imm8								Zd				
M																															

MOV <Zd>.<T>, <Pg>/Z, #<imm>{, <shift>}

is equivalent to

CPY <Zd>.<T>, <Pg>/Z, #<imm>{, <shift>}

and is always the preferred disassembly.

## Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<imm> Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #8

## Operation

The description of [CPY \(immediate, zeroing\)](#) gives the operational pseudocode for this instruction.

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (immediate, unpredicated)

Move signed immediate to vector elements (unpredicated)

Unconditionally broadcast the signed integer immediate into each element of the destination vector. This instruction is unpredicated.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This is an alias of [DUP \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [DUP \(immediate\)](#).
- The description of [DUP \(immediate\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	1	1	0	0	0	1	1	sh	imm8								Zd					

MOV <Zd>.<T>, #<imm>{, <shift>}

is equivalent to

DUP <Zd>.<T>, #<imm>{, <shift>}

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm> Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #8

Operation

The description of [DUP \(immediate\)](#) gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



# MOV (predicate, predicated, merging)

Move predicates (merging)

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register remain unmodified. Does not set the condition flags.

This is an alias of [SEL \(predicates\)](#). This means:

- The encodings in this description are named to match the encodings of [SEL \(predicates\)](#).
- The description of [SEL \(predicates\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	0	0	0	Pm			0		1	Pg			1		Pn			1		Pd			
S																															

MOV <Pd>.B, <Pg>/M, <Pn>.B

is equivalent to

SEL <Pd>.B, <Pg>, <Pn>.B, <Pd>.B

and is the preferred disassembly when Pd == Pm.

## Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

## Operation

The description of [SEL \(predicates\)](#) gives the operational pseudocode for this instruction.

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# MOV (predicate, predicated, zeroing)

Move predicates (zeroing)

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is an alias of [AND \(predicates\)](#). This means:

- The encodings in this description are named to match the encodings of [AND \(predicates\)](#).
- The description of [AND \(predicates\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	0	0	0	Pm				0	1	Pg				0	Pn				0	Pd			
S																															

MOV <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to

AND <Pd>.B, <Pg>/Z, <Pn>.B, <Pn>.B

and is the preferred disassembly when S == '0' && Pn == Pm.

## Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

## Operation

The description of [AND \(predicates\)](#) gives the operational pseudocode for this instruction.

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (scalar, predicated)

Move general-purpose register to vector elements (predicated)

Move the general-purpose scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of [CPY \(scalar\)](#). This means:

- The encodings in this description are named to match the encodings of [CPY \(scalar\)](#).
- The description of [CPY \(scalar\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	0	1	0	0	0	1	0	1	Pg			Rn			Zd							

MOV <Zd>.<T>, <Pg>/M, <R><n | SP>

is equivalent to

CPY <Zd>.<T>, <Pg>/M, <R><n | SP>

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<R> Is a width specifier, encoded in "size":

size	<R>
01	W
x0	W
11	X

<n|SP> Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

Operation

The description of [CPY \(scalar\)](#) gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.



This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (scalar, unpredicated)

Move general-purpose register to vector elements (unpredicated)

Unconditionally broadcast the general-purpose scalar source register into each element of the destination vector. This instruction is unpredicated.

This is an alias of [DUP \(scalar\)](#). This means:

- The encodings in this description are named to match the encodings of [DUP \(scalar\)](#).
- The description of [DUP \(scalar\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	0	0	0	0	0	0	0	1	1	1	0	Rn				Zd						

MOV <Zd>.<T>, <R><n | SP>

is equivalent to

[DUP](#) <Zd>.<T>, <R><n | SP>

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in "size":

size	<R>
01	W
x0	W
11	X

<n|SP> Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

Operation

The description of [DUP \(scalar\)](#) gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (SIMD&FP scalar, predicated)

Move SIMD&FP scalar register to vector elements (predicated)

Move the SIMD & floating-point scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of [CPY \(SIMD&FP scalar\)](#). This means:

- The encodings in this description are named to match the encodings of [CPY \(SIMD&FP scalar\)](#).
- The description of [CPY \(SIMD&FP scalar\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	0	0	0	0	0	1	0	0	Pg	Vn				Zd								

MOV <Zd>.<T>, <Pg>/M, <V><n>

is equivalent to

CPY <Zd>.<T>, <Pg>/M, <V><n>

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<V> Is a width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	D

<n> Is the number [0-31] of the source SIMD&FP register, encoded in the "Vn" field.

Operation

The description of [CPY \(SIMD&FP scalar\)](#) gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (SIMD&FP scalar, unpredicated)

Move indexed element or SIMD&FP scalar to vector (unpredicated)

Unconditionally broadcast the SIMD&FP scalar into each element of the destination vector. This instruction is unpredicated.

This is an alias of [DUP \(indexed\)](#). This means:

- The encodings in this description are named to match the encodings of [DUP \(indexed\)](#).
- The description of [DUP \(indexed\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	imm2	1	tsz				0				0	1	0	0	0	Zn				Zd				

MOV <Zd>.<T>, <Zn>.<T>[<imm>]

is equivalent to

DUP <Zd>.<T>, <Zn>.<T>[<imm>]

and is the preferred disassembly when BitCount(imm2:tsz) > 1.

MOV <Zd>.<T>, <V><n>

is equivalent to

DUP <Zd>.<T>, <Zn>.<T>[0]

and is the preferred disassembly when BitCount(imm2:tsz) == 1.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tsz":

tsz	<T>
00000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D
10000	Q

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<imm> Is the immediate index, in the range 0 to one less than the number of elements in 512 bits, encoded in "imm2:tsz".

<V> Is a width specifier, encoded in "tsz":

tsz	<V>
00000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D
10000	Q

<n> Is the number [0-31] of the source SIMD&FP register, encoded in the "Zn" field.

## Operation

The description of [DUP \(indexed\)](#) gives the operational pseudocode for this instruction.

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# MOV (vector, predicated)

Move vector elements (predicated)

Move elements from the source vector to the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of [SEL \(vectors\)](#). This means:

- The encodings in this description are named to match the encodings of [SEL \(vectors\)](#).
- The description of [SEL \(vectors\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	Zm				1	1	Pv			Zn				Zd								

MOV <Zd>.<T>, <Pv>/M, <Zn>.<T>

is equivalent to

SEL <Zd>.<T>, <Pv>, <Zn>.<T>, <Zd>.<T>

and is the preferred disassembly when **Zd == Zm**.

## Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pv> Is the name of the vector select predicate register, encoded in the "Pv" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

## Operation

The description of [SEL \(vectors\)](#) gives the operational pseudocode for this instruction.

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (vector, unpredicated)

Move vector register (unpredicated)

Move vector register. This instruction is unpredicated.

This is an alias of [ORR \(vectors, unpredicated\)](#). This means:

- The encodings in this description are named to match the encodings of [ORR \(vectors, unpredicated\)](#).
- The description of [ORR \(vectors, unpredicated\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	1	1	Zm				0	0	1	1	0	0	Zn				Zd						

MOV <Zd>.D, <Zn>.D

is equivalent to

ORR <Zd>.D, <Zn>.D, <Zn>.D

and is the preferred disassembly when **Zn == Zm**.

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

Operation

The description of [ORR \(vectors, unpredicated\)](#) gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



MOVPRFX (predicated)

Move prefix (predicated)

The predicated MOVPRFX instruction is a hint to hardware that the instruction may be combined with the destructive instruction which follows it in program order to create a single constructive operation. Since it is a hint it is also permitted to be implemented as a discrete vector copy, and the result of executing the pair of instructions with or without combining is identical. The choice of combined versus discrete operation may vary dynamically. Unless the combination of a constructive operation with merging predication is specifically required, it is strongly recommended that for performance reasons software should prefer to use the zeroing form of predicated MOVPRFX or the unpredicated MOVPRFX instruction.

Although the operation of the instruction is defined as a simple predicated vector copy, it is required that the prefixed instruction at PC+4 must be an SVE destructive binary or ternary instruction encoding, or a unary operation with merging predication, but excluding other MOVPRFX instructions. The prefixed instruction must specify the same predicate register, and have the same maximum element size (ignoring a fixed 64-bit "wide vector" operand), and the same destination vector as the MOVPRFX instruction. The prefixed instruction must not use the destination register in any other operand position, even if they have different names but refer to the same architectural register state. Any other use is UNPREDICTABLE.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	0	0	0	M	0	0	1	Pg			Zn				Zd						

MOVPRFX <Zd>.<T>, <Pg>/<ZM>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean merging = (M == '1');
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<ZM> Is the predication qualifier, encoded in "M":

M	<ZM>
0	Z
1	M

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) dest = Z[d, VL];
bits(VL) result;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand1, e, esize];
        Elem[result, e, esize] = element;
    elseif merging then
        Elem[result, e, esize] = Elem[dest, e, esize];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOVPRFX (unpredicated)

Move prefix (unpredicated)

The unpredicated MOVPRFX instruction is a hint to hardware that the instruction may be combined with the destructive instruction which follows it in program order to create a single constructive operation. Since it is a hint it is also permitted to be implemented as a discrete vector copy, and the result of executing the pair of instructions with or without combining is identical. The choice of combined versus discrete operation may vary dynamically. Although the operation of the instruction is defined as a simple unpredicated vector copy, it is required that the prefixed instruction at PC+4 must be an SVE destructive binary or ternary instruction encoding, or a unary operation with merging predication, but excluding other MOVPRFX instructions. The prefixed instruction must specify the same destination vector as the MOVPRFX instruction. The prefixed instruction must not use the destination register in any other operand position, even if they have different names but refer to the same architectural register state. Any other use is UNPREDICTABLE.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	1	1	1	Zn				Zd					

MOVPRFX <Zd>, <Zn>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(VL) result = Z[n, VL];
Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

MOVS (predicated)

Move predicates (zeroing), setting the condition flags

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is an alias of [ANDS](#). This means:

- The encodings in this description are named to match the encodings of [ANDS](#).
- The description of [ANDS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	1	0	0	Pm			0	1	Pg			0	Pn			0	Pd						
S																															

MOVS <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to

ANDS <Pd>.B, <Pg>/Z, <Pn>.B, <Pn>.B

and is the preferred disassembly when `S == '1' && Pn == Pm`.

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of [ANDS](#) gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

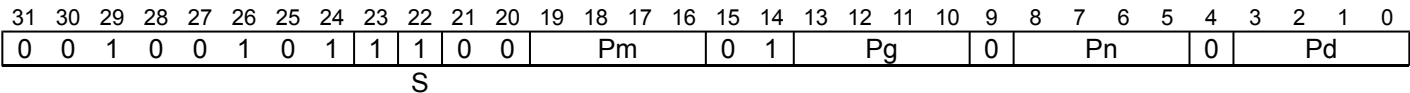
MOVS (unpredicated)

Move predicate (unpredicated), setting the condition flags

Read all elements from the source predicate and place in the destination predicate. This instruction is unpredicated. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is an alias of ORRS. This means:

- The encodings in this description are named to match the encodings of ORRS.
- The description of ORRS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



```
MOVS <Pd>.B, <Pn>.B
```

is equivalent to

```
ORRS <Pd>.B, <Pn>/Z, <Pn>.B, <Pn>.B
```

and is the preferred disassembly when S == '1' && Pn == Pm && Pm == Pg.

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of ORRS gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

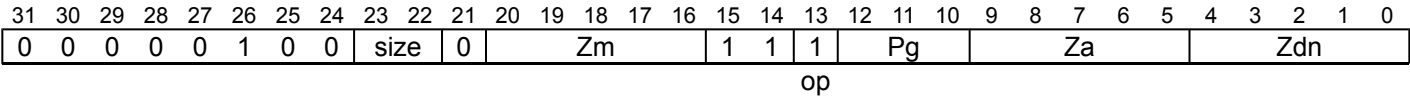
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

MSB

Multiply-subtract vectors (predicated), writing multiplicand [Zdn = Za - Zdn \* Zm]

Multiply the corresponding active elements of the first and second source vectors and subtract from elements of the third (addend) vector. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.



MSB <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean sub_op = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Za> Is the name of the third source scalable vector register, encoded in the "Za" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer product = element1 * element2;
    if sub_op then
      Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MUL (immediate)

Multiply by immediate (unpredicated)

Multiply by an immediate each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	1	0	0	0	0	1	1	0	imm8								Zdn					

```
MUL <Zdn>.<T>, <Zdn>.<T>, #<imm>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = SInt(imm8);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm> Is the signed immediate operand, in the range -128 to 127, encoded in the "imm8" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    Elem[result, e, esize] = (element1 * imm)<esize-1:0>;

Z[dn, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.





# MUL (indexed)

Multiply (indexed)

Multiply all integer elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The results are placed in the corresponding elements of the destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

It has encodings from 3 classes: [16-bit](#) , [32-bit](#) and [64-bit](#)

## 16-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	0	i3h	1	i3l	Zm			1	1	1	1	1	0	Zn				Zd						

MUL <Zd>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

## 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	0	1	i2	Zm			1	1	1	1	1	0	Zn				Zd						
size<1>size<0>																															

MUL <Zd>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

## 64-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	1	1	i1	Zm			1	1	1	1	1	0	Zn				Zd						
size<1>size<0>																															

MUL <Zd>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

## Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

- <Zm> For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
- For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.
- For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

## Operation

```

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, s, esize]);
    integer res = element1 * element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;

```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## MUL (vectors, predicated)

Multiply vectors (predicated)

Multiply active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	0	0	0	0	0	0	0	0	Pg	Zm			Zdn								
H																U															

**MUL** <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

### Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer product = element1 * element2;
        Elem[result, e, esize] = product<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```

### Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MUL (vectors, unpredicated)

Multiply vectors (unpredicated)

Multiply all elements of the first source vector by corresponding elements of the second source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	1	0	0	size	1	Zm						0	1	1	0	0	0	Zn						Zd				

```
MUL <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer product = element1 * element2;
    Elem[result, e, esize] = product<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

# NAND

## Bitwise NAND predicates

Bitwise NAND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	1	0	0	0	Pm				0	1	Pg				1	Pn				1	Pd			
S																															

NAND <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

## Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit element1 = PredicateElement(operand1, e, esize);
    bit element2 = PredicateElement(operand2, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, psize] = ZeroExtend(NOT(element1 AND element2), psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# NANDS

Bitwise NAND predicates, setting the condition flags

Bitwise NAND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	1	1	0	0		Pm			0	1		Pg		1			Pn		1			Pd		
																S															

NANDS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

## Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit element1 = PredicateElement(operand1, e, esize);
    bit element2 = PredicateElement(operand2, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, psize] = ZeroExtend(NOT(element1 AND element2), psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

NBSL

Bitwise inverted select

Selects bits from the first source vector where the corresponding bit in the third source vector is '1', and from the second source vector where the corresponding bit in the third source vector is '0'. The inverted result is placed destructively in the destination and first source vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	1	1					Zm	0	0	1	1	1	1				Zk				Zdn		

```
NBSL <Zdn>.D, <Zdn>.D, <Zm>.D, <Zk>.D
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);
```

Assembler Symbols

- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <Zk> Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[k, VL];

Z[dn, VL] = NOT((operand1 AND operand3) OR (operand2 AND NOT(operand3)));
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

NEG

Negate (predicated)

Negate the signed integer value in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	0	1	1	1	1	0	1	Pg													

NEG <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer element = SInt(Elem[operand, e, esize]);
        element = -element;
        Elem[result, e, esize] = element<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# NMATCH

Detect no matching elements, setting the condition flags

This instruction compares each active 8-bit or 16-bit character in the first source vector with all of the characters in the corresponding 128-bit segment of the second source vector. Where the first source element detects no matching characters in the second segment it places true in the corresponding element of the destination predicate, otherwise false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size	1	Zm						1	0	0	Pg			Zn				1	Pd				

NMATCH <Pd>.<T>, <Pg>/z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() then UNDEFINED;
if size IN {'lx'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Pd);
integer n = UInt(Zn);
integer m = UInt(Zm);
```

## Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size<0>":

size<0>	<T>
0	B
1	H

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer segmentbase = e - (e MOD eltspersegment);
        Elem[result, e, psize] = ZeroExtend('1', psize);
        bits(esize) element1 = Elem[operand1, e, esize];
        for i = segmentbase to (segmentbase + eltspersegment) - 1
            bits(esize) element2 = Elem[operand2, i, esize];
            if element1 == element2 then
                Elem[result, e, psize] = ZeroExtend('0', psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# NOR

## Bitwise NOR predicates

Bitwise NOR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	1	0	0	0	Pm				0	1	Pg				1	Pn				0	Pd			
S																															

NOR <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

## Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit element1 = PredicateElement(operand1, e, esize);
    bit element2 = PredicateElement(operand2, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, psize] = ZeroExtend(NOT(element1 OR element2), psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:



- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## NORS

Bitwise NOR predicates, setting the condition flags

Bitwise NOR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	1	1	0	0	Pm			0	1	Pg			1	Pn			0	Pd						
S																															

**NORS** <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

### Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit element1 = PredicateElement(operand1, e, esize);
    bit element2 = PredicateElement(operand2, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, psize] = ZeroExtend(NOT(element1 OR element2), psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

### Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# NOT (predicate)

Bitwise invert predicate

Bitwise invert each active element of the source predicate, and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is an alias of [EOR \(predicates\)](#). This means:

- The encodings in this description are named to match the encodings of [EOR \(predicates\)](#).
- The description of [EOR \(predicates\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	0	0	0	Pm				0	1	Pg				1	Pn				0	Pd			
S																															

NOT <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to

EOR <Pd>.B, <Pg>/Z, <Pn>.B, <Pg>.B

and is the preferred disassembly when Pm == Pg.

## Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

## Operation

The description of [EOR \(predicates\)](#) gives the operational pseudocode for this instruction.

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

NOT (vector)

Bitwise invert vector (predicated)

Bitwise invert each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	1	1	0	1	0	1	Pg				Zn				Zd					

NOT <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esome) element = Elem[operand, e, esize];
        Elem[result, e, esize] = NOT element;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

NOTS

Bitwise invert predicate, setting the condition flags

Bitwise invert each active element of the source predicate, and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is an alias of [EORS](#). This means:

- The encodings in this description are named to match the encodings of [EORS](#).
- The description of [EORS](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	1	0	0	1	0	1	0	1	0	0	Pm			0	1	Pg			1	Pn			0	Pd								
S																																	

NOTS [<Pd>.B](#), [<Pg>/Z](#), [<Pn>.B](#)

is equivalent to

[EORS](#) [<Pd>.B](#), [<Pg>/Z](#), [<Pn>.B](#), [<Pg>.B](#)

and is the preferred disassembly when [Pm == Pg](#).

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of [EORS](#) gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# ORN (immediate)

Bitwise inclusive OR with inverted immediate (unpredicated)

Bitwise inclusive OR an inverted immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This is a pseudo-instruction of [ORR \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [ORR \(immediate\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [ORR \(immediate\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	0	0	0	0	0	imm13														Zdn			

ORN <Zdn>.<T>, <Zdn>.<T>, #<const>

is equivalent to

ORR <Zdn>.<T>, <Zdn>.<T>, #(-<const> - 1)

## Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "imm13<12>:imm13<5:0>":

imm13<12>	imm13<5:0>	<T>
0	0xxxxx	S
0	10xxxx	H
0	110xxx	B
0	1110xx	B
0	11110x	B
0	111110	RESERVED
0	111111	RESERVED
1	xxxxxx	D

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

## Operation

The description of [ORR \(immediate\)](#) gives the operational pseudocode for this instruction.

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.





# ORN (predicates)

Bitwise inclusive OR inverted predicate

Bitwise inclusive OR inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	1	0	0	0	Pm				0	1	Pg				0	Pn				1	Pd			
S																															

ORN <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

## Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit element1 = PredicateElement(operand1, e, esize);
    bit element2 = PredicateElement(operand2, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, psize] = ZeroExtend(element1 OR (NOT element2), psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

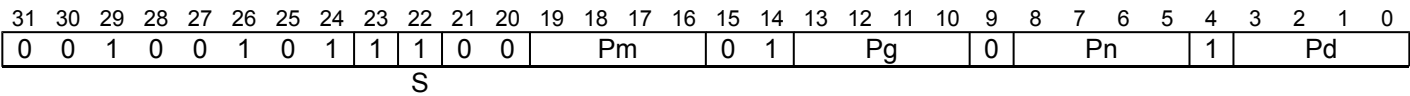
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# ORNS

Bitwise inclusive OR inverted predicate, setting the condition flags

Bitwise inclusive OR inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



ORNS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

## Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit element1 = PredicateElement(operand1, e, esize);
    bit element2 = PredicateElement(operand2, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, psize] = ZeroExtend(element1 OR (NOT element2), psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORQV

Bitwise inclusive OR reduction of quadword vector segments

Bitwise inclusive OR of the same element numbers from each 128-bit source vector segment, placing each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as all zeros.

SVE2  
(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	1	0	0	0	0	1	Pg			Zn				Vd						

ORQV <Vd>.<T>, <Pg>, <Zn>.<Tb>

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<Vd> Is the name of the destination SIMD&FP register, encoded in the "Vd" field.

<T> Is an arrangement specifier, encoded in "size":

size	<T>
00	16B
01	8H
10	4S
11	2D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	B
01	H
10	S
11	D

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elemperssegment = 128 DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(128) result = Zeros(128);
bits(128) stmp = Zeros(128);

bits(esize) dtmp;

for e = 0 to elemperssegment-1
    dtmp = Zeros(esize);
    for s = 0 to segments-1
        if ActivePredicateElement(mask, s * elemperssegment + e, esize) then
            stmp = Elem[operand, s, 128];
            dtmp = dtmp OR Elem[stmp, e, esize];
        Elem[result, e, esize] = dtmp<esize-1:0>;

V[d, 128] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# ORR (immediate)

Bitwise inclusive OR with immediate (unpredicated)

Bitwise inclusive OR an immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction [ORN \(immediate\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	0	0	0	0	0	imm13														Zdn			

ORR <Zdn>.<T>, <Zdn>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer dn = UInt(Zdn);
bits(64) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE, 64);
```

## Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "imm13<12>:imm13<5:0>":

imm13<12>	imm13<5:0>	<T>
0	0xxxxxx	S
0	10xxxxx	H
0	110xxxx	B
0	1110xxx	B
0	11110xx	B
0	111110	RESERVED
0	111111	RESERVED
1	xxxxxxx	D

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 64;
bits(VL) operand = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(64) element1 = Elem[operand, e, 64];
    Elem[result, e, 64] = element1 OR imm;

Z[dn, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

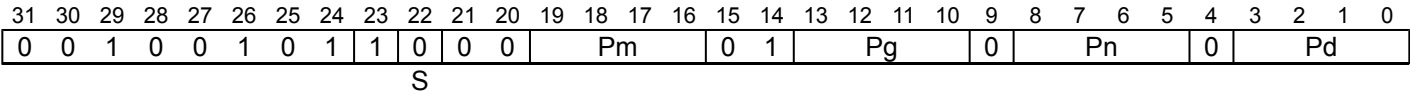
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORR (predicates)

Bitwise inclusive OR predicates

Bitwise inclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This instruction is used by the alias [MOV](#).



```
ORR <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
<a href="#">MOV</a>	S == '0' && Pn == Pm && Pm == Pg

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit element1 = PredicateElement(operand1, e, esize);
    bit element2 = PredicateElement(operand2, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, psize] = ZeroExtend(element1 OR element2, psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# ORR (vectors, predicated)

Bitwise inclusive OR vectors (predicated)

Bitwise inclusive OR active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	0	0	0	0	0	0	0	Pg												

ORR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

## Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = element1 OR element2;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORR (vectors, unpredicated)

Bitwise inclusive OR vectors (unpredicated)

Bitwise inclusive OR all elements of the second source vector with corresponding elements of the first source vector and place the first in the corresponding elements of the destination vector. This instruction is unpredicated.  
This instruction is used by the alias [MOV \(vector, unpredicated\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	1	1	Zm					0	0	1	1	0	0	Zn					Zd				

ORR <Zd>.D, <Zn>.D, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Alias Conditions

Alias	Is preferred when
<a href="#">MOV (vector, unpredicated)</a>	Zn == Zm

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];

Z[d, VL] = operand1 OR operand2;
```

Operational information

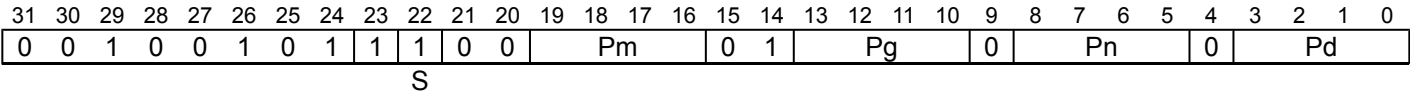
- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

ORRS

Bitwise inclusive OR predicates, setting the condition flags

Bitwise inclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the alias [MOVS \(unpredicated\)](#).



ORRS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
<a href="#">MOVS (unpredicated)</a>	S == '1' && Pn == Pm && Pm == Pg

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit element1 = PredicateElement(operand1, e, esize);
    bit element2 = PredicateElement(operand2, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, psize] = ZeroExtend(element1 OR element2, psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



ORV

Bitwise inclusive OR reduction to scalar

Bitwise inclusive OR horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as zero.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	0	0	0	0	0	0	1	Pg												

ORV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) result = Zeros(esize);

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        result = result OR Elem[operand, e, esize];

V[d, esize] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PEXT (predicate pair)

Set pair of predicates from predicate-as-counter

Expands the source predicate-as-counter into a four-predicate wide mask and copies two quarters of it into the destination predicate registers.

SVE2  
(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	0	0	0	0	0	0	0	1	1	1	0	1	0	i1	PNn		1	Pd				

```
PEXT { <Pd1>.<T>, <Pd2>.<T> }, <PNn>[<imm>]
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt('1':PNn);
integer d0 = UInt(Pd);
integer d1 = (UInt(Pd) + 1) MOD 16;
integer part = UInt(i1);
```

Assembler Symbols

<Pd1> Is the name of the first destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pd2> Is the name of the second destination scalable predicate register, encoded in the "Pd" field.

<PNn> Is the name of the first source scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNn" field.

<imm> Is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) pred = P[n, PL];
bits(PL*4) mask = CounterToPredicate(pred<15:0>, PL*4);
bits(PL) result0;
bits(PL) result1;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit pbit = PredicateElement(mask, part * 2 * elements + e, esize);
    Elem[result0, e, psize] = ZeroExtend(pbit, psize);

for e = 0 to elements-1
    bit pbit = PredicateElement(mask, part * 2 * elements + elements + e, esize);
    Elem[result1, e, psize] = ZeroExtend(pbit, psize);

P[d0, PL] = result0;
P[d1, PL] = result1;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

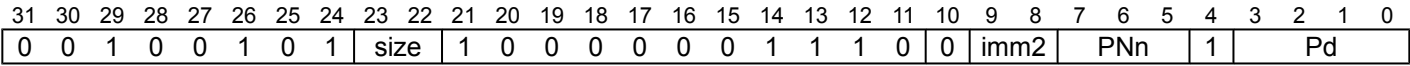
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PEXT (predicate)

Set predicate from predicate-as-counter

Expands the source predicate-as-counter into a four-predicate wide mask and copies one quarter of it into the destination predicate register.

SVE2
(FEAT\_SVE2p1)



PEXT <Pd>.<T>, <PNn>[<imm>]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt('1':PNn);
integer d = UInt(Pd);
integer part = UInt(imm2);
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<PNn> Is the name of the first source scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNn" field.

<imm> Is the element index, in the range 0 to 3, encoded in the "imm2" field.

Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) pred = P[n, PL];
bits(PL*4) mask = CounterToPredicate(pred<15:0>, PL*4);
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit pbit = PredicateElement(mask, part * elements + e, esize);
    Elem[result, e, psize] = ZeroExtend(pbit, psize);

P[d, PL] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PFALSE

Set all predicate elements to false

Set all elements in the destination predicate to false.

For programmer convenience, an assembler must also accept predicate-as-counter register name for the destination predicate register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	0	0	1	1	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0				Pd
																S															

PFALSE <Pd>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer d = UInt(Pd);
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
P[d, PL] = Zeros(PL);
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PFIRST

Set the first active predicate element to true

Sets the first active element in the destination predicate to true, otherwise elements from the source predicate are passed through unchanged. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	1	0	1	1	0	0	0	1	1	0	0	0	0	0	Pg			0	Pdn				
S																															

```
PFIRST <Pdn>.B, <Pg>, <Pdn>.B
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer dn = UInt(Pdn);
```

Assembler Symbols

- <Pdn> Is the name of the source and destination scalable predicate register, encoded in the "Pdn" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) result = P[dn, PL];
integer first = -1;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) && first == -1 then
        first = e;

if first >= 0 then
    Elem[result, first, psize] = ZeroExtend('1', psize);

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[dn, PL] = result;
```

Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.



## PMOV (to predicate)

Move predicate from vector

Copy a packed bitmap, where bit value 0b1 represents TRUE and bit value 0b0 represents FALSE, from part of a source vector register to elements of a destination SVE predicate register.

Because the number of bits in an SVE predicate element scales with the vector element size, the behavior varies according to the specified element size.

- When the predicate element specifier is.B, each bit [N] from the least-significant VL/8 bits in the source vector register is copied to bit [N] of the destination predicate register. The immediate index, if specified, must be 0.
- When the predicate element specifier is.H, each bit [N] within the indexed block of VL/16 bits in the source vector register is copied to bit [N\*2] of the destination predicate register, and the other bits in the predicate are set to zero. The immediate index is in the range 0 to 1, inclusive.
- When the predicate elements specifier is.S, each bit [N] within the indexed block of VL/32 bits in the source vector register is copied to bit [N\*4] of the destination predicate register, and the other bits in the predicate are set to zero. The immediate index is in the range 0 to 3, inclusive.
- When the predicate element specifier is.D, each bit [N] within the indexed block of VL/64 bits in the source vector register is copied to bit [N\*8] of the destination predicate register, and the other bits in the predicate are set to zero. The immediate index is in the range 0 to 7, inclusive.

The immediate index is optional, defaulting to 0 if omitted.

It has encodings from 4 classes: [Byte](#) , [Doubleword](#) , [Halfword](#) and [Word](#)

### Byte

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	0	1	0	1	0	1	0	0	0	1	1	1	0	Zn				0	Pd				

PMOV <Pd>.B, <Zn>

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Pd);
constant integer esize = 8;
constant integer imm = 0;
```

### Doubleword

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	1	i3h	1	0	1	i3l	0	0	0	1	1	1	0	Zn				0	Pd					

PMOV <Pd>.D, <Zn>[<imm>]

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Pd);
constant integer esize = 64;
constant integer imm = UInt(i3h:i3l);
```

### Halfword

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	0	1	0	1	1	i1	0	0	0	1	1	1	0	Zn				0	Pd				

**PMOV <Pd>.H, <Zn>[<imm>]**

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Pd);
constant integer esize = 16;
constant integer imm = UInt(i1);
```

**Word**  
**(FEAT\_SVE2p1)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	1	1	0	1	i2		0	0	0	1	1	1	0	Zn				0	Pd				

**PMOV <Pd>.S, <Zn>[<imm>]**

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Pd);
constant integer esize = 32;
constant integer imm = UInt(i2);
```

**Assembler Symbols**

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
- <imm> For the doubleword variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the halfword variant: is the element index, in the range 0 to 1, encoded in the "i1" field.  
For the word variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand = Z[n, VL];
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    Elem[result, e, psize] = ZeroExtend(operand<(elements * imm) + e>, psize);

P[d, PL] = result;
```

# PMOV (to vector)

Move predicate to vector

Copy the source SVE predicate register elements into the destination vector register as a packed bitmap with one bit per predicate element, where bit value 0b1 represents a TRUE predicate element, and bit value 0b0 represents a FALSE predicate element.

Because the number of bits in an SVE predicate element scales with the the vector element size, the behavior varies according to the specified element size.

- When the predicate element specifier is.B, every bit in the predicate register is copied to the least-significant VL/8 bits of the destination vector register. The immediate index, if specified, must be 0.
- When the predicate element specifier is.H, every second bit in the predicate register is copied to the indexed block of VL/16 bits in the destination vector register, where the immediate index is in the range 0 to 1, inclusive.
- When the predicate element specifier is.S, every fourth bit in the predicate register is copied to the indexed block of VL/32 bits in the destination vector register, where the immediate index is in the range 0 to 3, inclusive.
- When the predicate element specifier is.D, every eighth bit in the predicate register is copied to the indexed block of VL/64 bits in the destination vector register, where the immediate index is in the range 0 to 7, inclusive.

The immediate index is optional, defaulting to 0 if omitted. When the index is zero, the instruction writes zeroes to the most significant VL-(VL/esize) bits of the destination vector register. When a non-zero index is specified, the packed bitmap is inserted into the destination vector register, and the unindexed blocks remain unchanged.

It has encodings from 4 classes: [Byte](#) , [Doubleword](#) , [Halfword](#) and [Word](#)

## Byte (FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	0	1	0	1	0	1	1	0	0	1	1	1	0	0	Pn				Zd				

PMOV <Zd> , <Pn>.B

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
integer n = UInt(Pn);
integer d = UInt(Zd);
constant integer esize = 8;
constant integer imm = 0;
```

## Doubleword (FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	1	i3h	1	0	1	i3l	1	0	0	1	1	1	0	0	Pn				Zd					

PMOV <Zd>[<imm>] , <Pn>.D

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
integer n = UInt(Pn);
integer d = UInt(Zd);
constant integer esize = 64;
constant integer imm = UInt(i3h:i3l);
```

## Halfword (FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	0	1	0	1	1	i1	1	0	0	1	1	1	0	0	Pn				Zd				

**PMOV <Zd>[<imm>], <Pn>.H**

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
integer n = UInt(Pn);
integer d = UInt(Zd);
constant integer esize = 16;
constant integer imm = UInt(i1);
```

**Word**  
**(FEAT\_SVE2p1)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	1	1	0	1	i2	1	0	0	1	1	1	0	0	Pn				Zd					

**PMOV <Zd>[<imm>], <Pn>.S**

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
integer n = UInt(Pn);
integer d = UInt(Zd);
constant integer esize = 32;
constant integer imm = UInt(i2);
```

**Assembler Symbols**

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <imm> For the doubleword variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the halfword variant: is the element index, in the range 0 to 1, encoded in the "i1" field.  
For the word variant: is the element index, in the range 0 to 3, encoded in the "i2" field.
- <Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) operand = P[n, PL];
bits(VL) result;

if imm == 0 then
    result = Zeros(VL);
else
    result = Z[d, VL];

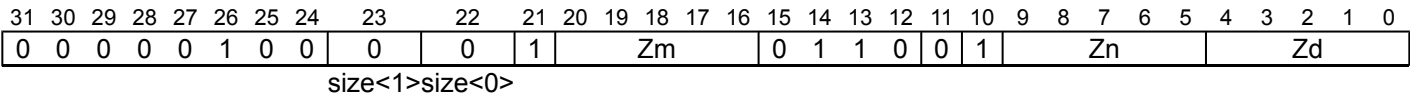
for e = 0 to elements-1
    result<(elements * imm) + e> = PredicateElement(operand, e, esize);

Z[d, VL] = result;
```

PMUL

Polynomial multiply vectors (unpredicated)

Polynomial multiply over [0, 1] all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.



```
PMUL <Zd>.B, <Zn>.B, <Zm>.B
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = PolynomialMult(element1, element2)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

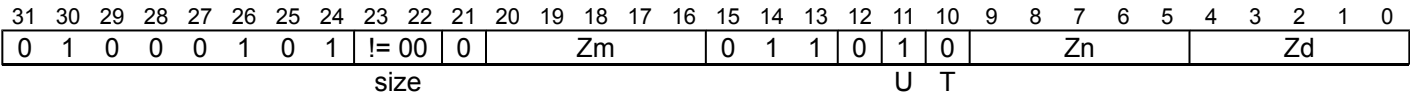
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

PMULLB

Polynomial multiply long (bottom)

Polynomial multiply over [0, 1] the corresponding even-numbered elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated. ID\_AA64ZFR0\_EL1.AES indicates whether the 128-bit element variant is implemented. The 128-bit element variant is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled. It has encodings from 2 classes: [16-bit or 64-bit elements](#) and [128-bit element](#)

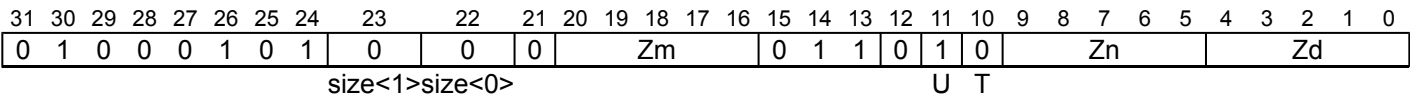
16-bit or 64-bit elements



PMULLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size<0> == '0' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

128-bit element  
(FEAT\_SVE\_PMULL128)



PMULLB <Zd>.Q, <Zn>.D, <Zm>.D

```
if !HaveSVE2PMULL128() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size<1>":

size<1>	<T>
0	H
1	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size<1>":

size<1>	<Tb>
0	B
1	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
if esize < 128 then CheckSVEEnabled\(\); else CheckNonStreamingSVEEnabled\(\);
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize DIV 2) element1 = Elem[operand1, 2*e + 0, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, 2*e + 0, esize DIV 2];
    Elem[result, e, esize] = PolynomialMult(element1, element2);

Z[d, VL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

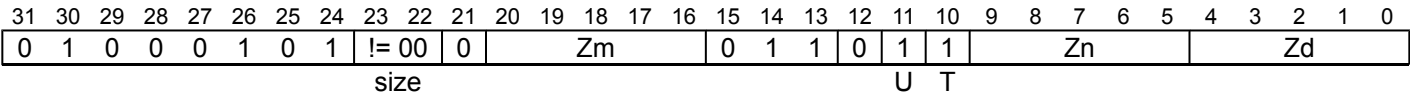
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PMULLT

Polynomial multiply long (top)

Polynomial multiply over [0, 1] the corresponding odd-numbered elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated. ID\_AA64ZFR0\_EL1.AES indicates whether the 128-bit element variant is implemented. The 128-bit element variant is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled. It has encodings from 2 classes: [16-bit or 64-bit elements](#) and [128-bit element](#)

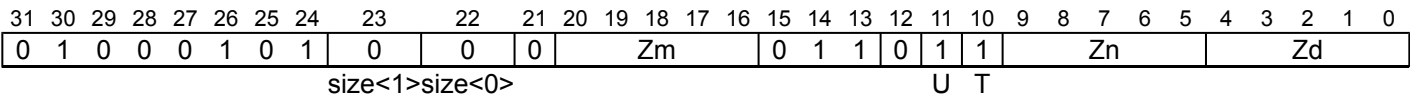
16-bit or 64-bit elements



```
PMULLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size<0> == '0' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

128-bit element  
(FEAT\_SVE\_PMULL128)



```
PMULLT <Zd>.Q, <Zn>.D, <Zm>.D
```

```
if !HaveSVE2PMULL128() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size<1>":

size<1>	<T>
0	H
1	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size<1>":

size<1>	<Tb>
0	B
1	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.



## Operation

```
if esize < 128 then CheckSVEEnabled\(\); else CheckNonStreamingSVEEnabled\(\);
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize DIV 2) element1 = Elem[operand1, 2*e + 1, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, 2*e + 1, esize DIV 2];
    Elem[result, e, esize] = PolynomialMult(element1, element2);

Z[d, VL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PNEXT

Find next active predicate

An instruction used to construct a loop which iterates over all true elements in the vector select predicate register. If all elements in the first source predicate register are false it determines the first true element in the vector select predicate register; otherwise it determines the next true element in the vector select predicate register that follows the last true element in the first source predicate register. All elements of the destination predicate register are set to false, except the element corresponding to the determined vector select element, if any, which is set to true. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	0	1	1	0	0	1	1	1	0	0	0	1	0	0	Pv	0					Pdn		

PNEXT <Pdn>.<T>, <Pv>, <Pdn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer v = UInt(Pv);
integer dn = UInt(Pdn);
```

Assembler Symbols

<Pdn> Is the name of the first source and destination scalable predicate register, encoded in the "Pdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pv> Is the name of the vector select predicate register, encoded in the "Pv" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[v, PL];
bits(PL) operand = P[dn, PL];
bits(PL) result;
constant integer psize = esize DIV 8;

integer next = LastActiveElement(operand, esize) + 1;

while next < elements && (!ActivePredicateElement(mask, next, esize)) do
    next = next + 1;

result = Zeros(PL);
if next < elements then
    Elem[result, next, psize] = ZeroExtend('1', psize);

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[dn, PL] = result;
```

Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

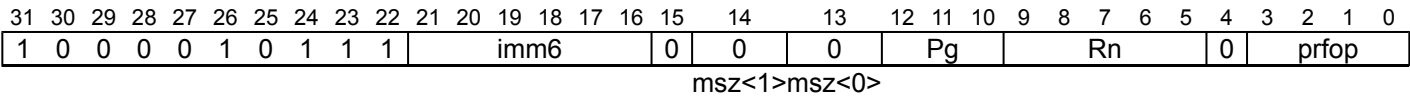


PRFB (scalar plus immediate)

Contiguous prefetch bytes (immediate index)

Contiguous prefetch of byte elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.



PRFB <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;
integer offset = SInt(imm6);
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in "prfop":

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(PL) mask = P[g, PL];  
bits(64) base;  
  
if AnyActiveElement(mask, esize) then  
    base = if n == 31 then SP[] else X[n, 64];  
  
for e = 0 to elements-1  
    if ActivePredicateElement(mask, e, esize) then  
        integer eoff = (offset * elements) + e;  
        bits(64) addr = base + (eoff << scale);  
        Hint Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

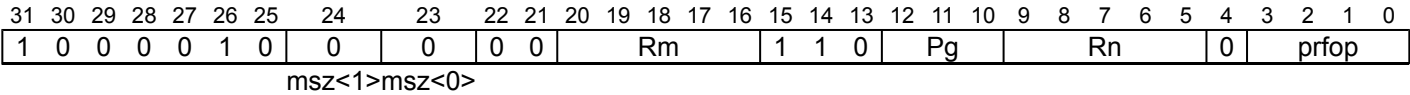
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PRFB (scalar plus scalar)

Contiguous prefetch bytes (scalar index)

Contiguous prefetch of byte elements from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The predicate may be used to suppress prefetches from unwanted addresses.



PRFB <prfop>, <Pg>, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(PL) mask = P[g, PL];  
bits(64) base;  
bits(64) offset;  
  
if AnyActiveElement(mask, esize) then  
    base = if n == 31 then SP[] else X[n, 64];  
    offset = X[m, 64];  
  
for e = 0 to elements-1  
    if ActivePredicateElement(mask, e, esize) then  
        integer eoff = UInt(offset) + e;  
        bits(64) addr = base + (eoff << scale);  
        Hint\_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## PRFB (scalar plus vector)

Gather prefetch bytes (scalar plus vector)

Gather prefetch of bytes from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 3 classes: [32-bit scaled offset](#) , [32-bit unpacked scaled offset](#) and [64-bit scaled offset](#)

### 32-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	0	xs	1			Zm		0	0	0		Pg					Rn		0			prfop		
																msz<1>msz<0>															

**PRFB** <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 0;
```

### 32-bit unpacked scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	xs	1			Zm		0	0	0		Pg					Rn		0			prfop		
																msz<1>msz<0>															

**PRFB** <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 0;
```

### 64-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	1	1	Zm			1	0	0	Pg			Rn			0	prfop							
																msz<1>msz<0>															



PRFB <prfop>, <Pg>, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```



PRFB (vector plus immediate)

Gather prefetch bytes (vector plus immediate)

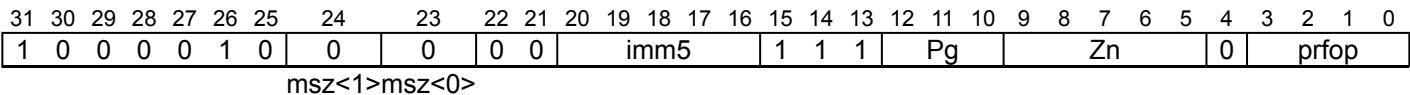
Gather prefetch of bytes from the active memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

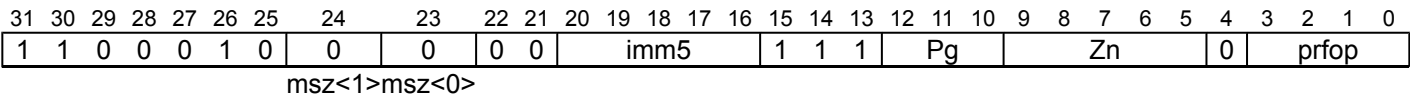
32-bit element



PRFB <prfop>, <Pg>, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;
integer offset = UInt(imm5);
```

64-bit element



PRFB <prfop>, <Pg>, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;
integer offset = UInt(imm5);
```

Assembler Symbols

<prfop>      Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

## Operation

```

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

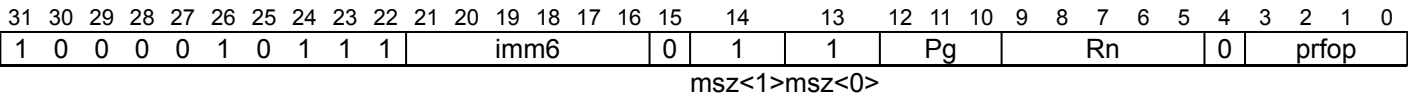
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PRFD (scalar plus immediate)

Contiguous prefetch doublewords (immediate index)

Contiguous prefetch of doubleword elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.



PRFD <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;
integer offset = SInt(imm6);
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(PL) mask = P[g, PL];  
bits(64) base;  
  
if AnyActiveElement(mask, esize) then  
    base = if n == 31 then SP[] else X[n, 64];  
  
for e = 0 to elements-1  
    if ActivePredicateElement(mask, e, esize) then  
        integer eoff = (offset * elements) + e;  
        bits(64) addr = base + (eoff << scale);  
        Hint Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

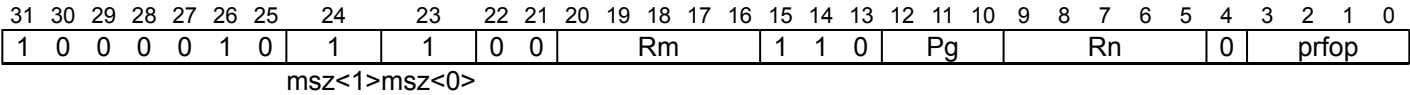
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PRFD (scalar plus scalar)

Contiguous prefetch doublewords (scalar index)

Contiguous prefetch of doubleword elements from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The predicate may be used to suppress prefetches from unwanted addresses.



PRFD <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(PL) mask = P[g, PL];  
bits(64) base;  
bits(64) offset;  
  
if AnyActiveElement(mask, esize) then  
    base = if n == 31 then SP[] else X[n, 64];  
    offset = X[m, 64];  
  
for e = 0 to elements-1  
    if ActivePredicateElement(mask, e, esize) then  
        integer eoff = UInt(offset) + e;  
        bits(64) addr = base + (eoff << scale);  
        Hint\_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## PRFD (scalar plus vector)

Gather prefetch doublewords (scalar plus vector)

Gather prefetch of doublewords from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then multiplied by 8. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 3 classes: [32-bit scaled offset](#) , [32-bit unpacked scaled offset](#) and [64-bit scaled offset](#)

### 32-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	0	xs	1			Zm		0	1	1		Pg					Rn		0			prfop		
																msz<1>msz<0>															

**PRFD** <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #3]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 3;
```

### 32-bit unpacked scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	xs	1			Zm		0	1	1		Pg					Rn		0			prfop		
																msz<1>msz<0>															

**PRFD** <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #3]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 3;
```

### 64-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	1	1	Zm			1	1	1	Pg			Rn			0	prfop							
																msz<1>msz<0>															

PRFD <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, LSL #3]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 3;
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```



PRFD (vector plus immediate)

Gather prefetch doublewords (vector plus immediate)

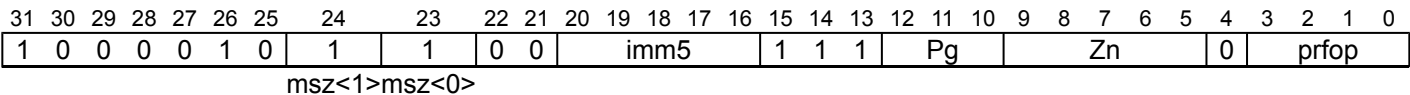
Gather prefetch of doublewords from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

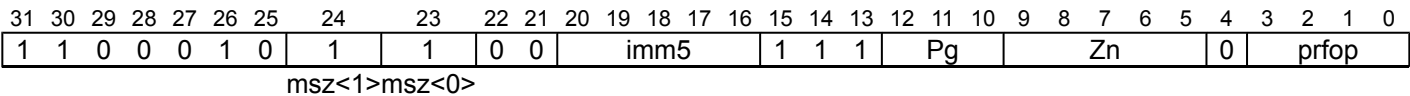
32-bit element



PRFD <prfop>, <Pg>, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;
integer offset = UInt(imm5);
```

64-bit element



PRFD <prfop>, <Pg>, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;
integer offset = UInt(imm5);
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

## Operation

```

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

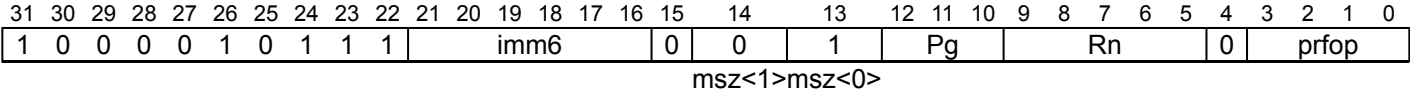
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PRFH (scalar plus immediate)

Contiguous prefetch halfwords (immediate index)

Contiguous prefetch of halfword elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.



```
PRFH <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;
integer offset = SInt(imm6);
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(PL) mask = P[g, PL];  
bits(64) base;  
  
if AnyActiveElement(mask, esize) then  
    base = if n == 31 then SP[] else X[n, 64];  
  
for e = 0 to elements-1  
    if ActivePredicateElement(mask, e, esize) then  
        integer eoff = (offset * elements) + e;  
        bits(64) addr = base + (eoff << scale);  
        Hint Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PRFH (scalar plus scalar)

Contiguous prefetch halfwords (scalar index)

Contiguous prefetch of halfword elements from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The predicate may be used to suppress prefetches from unwanted addresses.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	1	0	0	Rm				1	1	0	Pg			Rn				0	prfop					
msz<1>msz<0>																															

PRFH <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(64) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + (eoff << scale);
        Hint\_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## PRFH (scalar plus vector)

Gather prefetch halfwords (scalar plus vector)

Gather prefetch of halfwords from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then multiplied by 2. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 3 classes: [32-bit scaled offset](#) , [32-bit unpacked scaled offset](#) and [64-bit scaled offset](#)

### 32-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1 0 0 0 0 1 0 0 0									xs 1		Zm				0 0		1		Pg			Rn				0		prfop			
msz<1>msz<0>																															

**PRFH** <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 1;
```

### 32-bit unpacked scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	xs	1			Zm		0	0	1		Pg					Rn		0			prfop		
																msz<1>msz<0>															

**PRFH** <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 1;
```

### 64-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	1	1	Zm		1	0	1	Pg		Rn		0	prfop										
																msz<1>msz<0>															

PRFH <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, LSL #1]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 1;
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```



PRFH (vector plus immediate)

Gather prefetch halfwords (vector plus immediate)

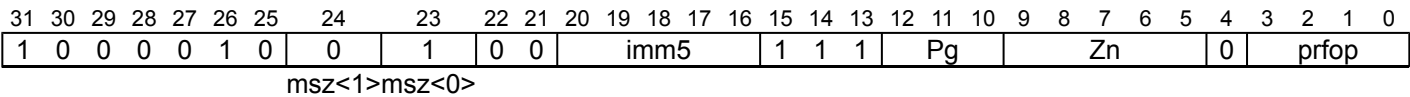
Gather prefetch of halfwords from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

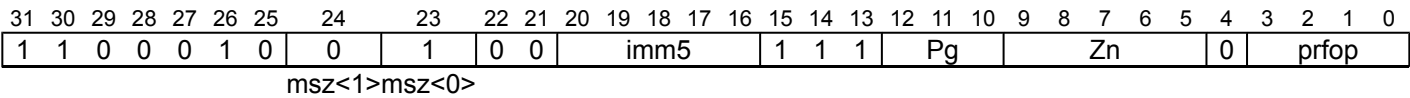
32-bit element



```
PRFH <prfop>, <Pg>, [<Zn>.S{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;
integer offset = UInt(imm5);
```

64-bit element



```
PRFH <prfop>, <Pg>, [<Zn>.D{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;
integer offset = UInt(imm5);
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

## Operation

```

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

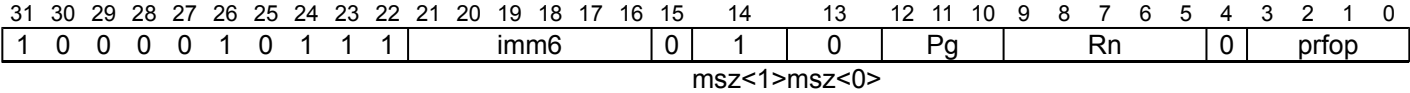
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PRFW (scalar plus immediate)

Contiguous prefetch words (immediate index)

Contiguous prefetch of word elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.



```
PRFW <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;
integer offset = SInt(imm6);
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(PL) mask = P[g, PL];  
bits(64) base;  
  
if AnyActiveElement(mask, esize) then  
    base = if n == 31 then SP[] else X[n, 64];  
  
for e = 0 to elements-1  
    if ActivePredicateElement(mask, e, esize) then  
        integer eoff = (offset * elements) + e;  
        bits(64) addr = base + (eoff << scale);  
        Hint Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

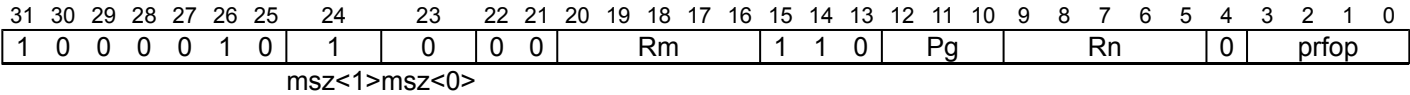


PRFW (scalar plus scalar)

Contiguous prefetch words (scalar index)

Contiguous prefetch of word elements from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The predicate may be used to suppress prefetches from unwanted addresses.



PRFW <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(64) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + (eoff << scale);
        Hint\_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## PRFW (scalar plus vector)

Gather prefetch words (scalar plus vector)

Gather prefetch of words from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then multiplied by 4. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 3 classes: [32-bit scaled offset](#) , [32-bit unpacked scaled offset](#) and [64-bit scaled offset](#)

### 32-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	0	xs	1			Zm		0	1	0		Pg					Rn		0			prfop		
																msz<1>msz<0>															

**PRFW** <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 2;
```

### 32-bit unpacked scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	xs	1			Zm		0	1	0		Pg					Rn		0			prfop		
																msz<1>msz<0>															

**PRFW** <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 2;
```

### 64-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	1	1	Zm		1	1	0	Pg		Rn		0	prfop										
																msz<1>msz<0>															

PRFW <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, LSL #2]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 2;
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

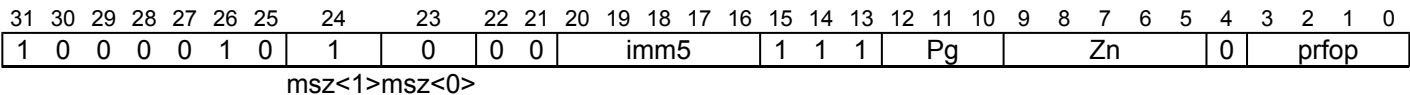


PRFW (vector plus immediate)

Gather prefetch words (vector plus immediate)

Gather prefetch of words from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive addresses are not prefetched from memory. The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal). This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled. It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

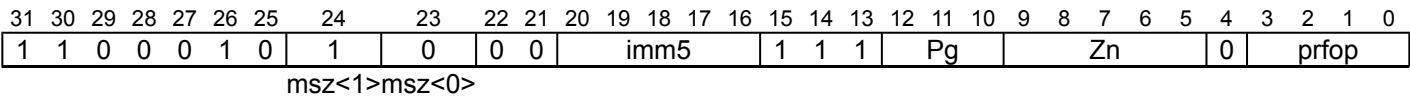
32-bit element



```
PRFW <prfop>, <Pg>, [<Zn>.S{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;
integer offset = UInt(imm5);
```

64-bit element



```
PRFW <prfop>, <Pg>, [<Zn>.D{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;
integer offset = UInt(imm5);
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

## Operation

```

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

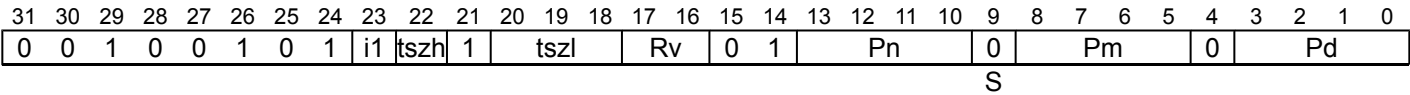
PSEL

Predicate select between predicate register or all-false

If the indexed element of the second source predicate is true, place the contents of the first source predicate register into the destination predicate register, otherwise set the destination predicate to all-false. The indexed element is determined by the sum of a general-purpose index register and an immediate, modulo the number of elements. Does not set the condition flags.

For programmer convenience, an assembler must also accept predicate-as-counter register names for the destination predicate register and the first source predicate register.

SVE2  
(FEAT\_SVE2p1)



```
PSEL <Pd>, <Pn>, <Pm>.<T>[<Wv>, <imm>]
```

```
if !HaveSME() && !HaveSVE2p1() then UNDEFINED;
bits(5) imm5 = i1:tszh:tszl;
integer esize;
integer imm;
case tszh:tszl of
  when '0000' UNDEFINED;
  when '1000' esize = 64;  imm = UInt(imm5<4>);
  when 'x100' esize = 32;  imm = UInt(imm5<4:3>);
  when 'xx10' esize = 16;  imm = UInt(imm5<4:2>);
  when 'xxx1' esize = 8;   imm = UInt(imm5<4:1>);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer v = UInt('011':Rv);
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.
- <T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	000	RESERVED
x	xx1	B
x	x10	H
x	100	S
1	000	D

- <Wv> Is the 32-bit name of the vector select register W12-W15, encoded in the "Rv" field.
- <imm> Is the element index, in the range 0 to one less than the number of vector elements in a 128-bit vector register, encoded in "i1:tszh:tszl".



## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(PL) operand1 = P[n, PL];  
bits(PL) operand2 = P[m, PL];  
bits(32) idx = X[v, 32];  
integer element = (UInt(idx) + imm) MOD elements;  
bits(PL) result;  
  
if ActivePredicateElement(operand2, element, esize) then  
    result = operand1;  
else  
    result = Zeros(PL);  
  
P[d, PL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PTEST

Set condition flags for predicate

Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate source register, and the V flag to zero.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	1	0	1	0	0	0	0	1	1	Pg			0	Pn			0	0	0	0	0	0	0
S																															

PTEST <Pg>, <Pn>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
```

Assembler Symbols

- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(PL) result = P[n, PL];

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

PTRUE (predicate as counter)

Initialise predicate-as-counter to all active

Set the destination predicate as all-active elements, using the predicate-as-counter encoding.

SVE2  
(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	1	0				PNd

PTRUE <PNd>.<T>

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer d = UInt('1':PNd);
```

Assembler Symbols

- <PNd> Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNd" field.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) result = EncodePredCount(esome, elements, elements, FALSE, PL);
P[d, PL] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

PTRUE (predicate)

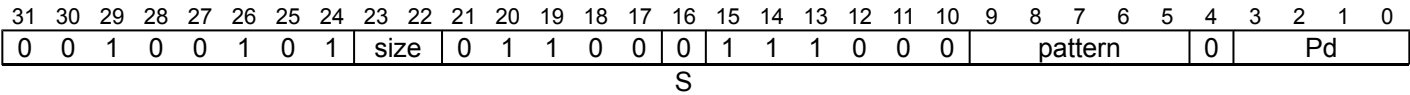
Initialise predicate from named constraint

Set elements of the destination predicate to true if the element number satisfies the named predicate constraint, or to false otherwise. If the constraint specifies more elements than are available at the current vector length then all elements of the destination predicate are set to false.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception. Does not set the condition flags.



```
PTRUE <Pd>.<T>{, <pattern>}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer d = UInt(Pd);
boolean setflags = FALSE;
bits(5) pat = pattern;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

## Operation

```

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit pbit = if e < count then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(result, result, esize);
P[d, PL] = result;

```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PTRUES

Initialise predicate from named constraint and set the condition flags

Set elements of the destination predicate to true if the element number satisfies the named predicate constraint, or to false otherwise. If the constraint specifies more elements than are available at the current vector length then all elements of the destination predicate are set to false.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size		0	1	1	0	0	1	1	1	1	0	0	0	pattern					0	Pd			
S																															

PTRUES <Pd>.<T>{, <pattern>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer d = UInt(Pd);
boolean setflags = TRUE;
bits(5) pat = pattern;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

## Operation

```

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit pbit = if e < count then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(result, result, esize);
P[d, PL] = result;

```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

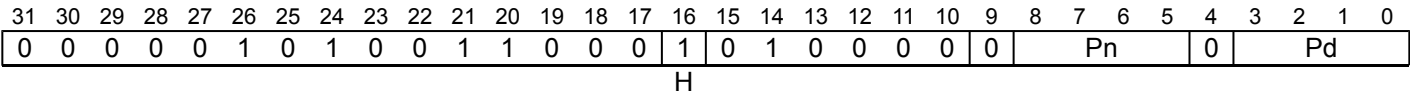
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# PUNPKHI, PUNPKLO

Unpack and widen half of predicate

Unpack elements from the lowest or highest half of the source predicate and place in elements of twice their size within the destination predicate. This instruction is unpredicated.  
It has encodings from 2 classes: [High half](#) and [Low half](#)

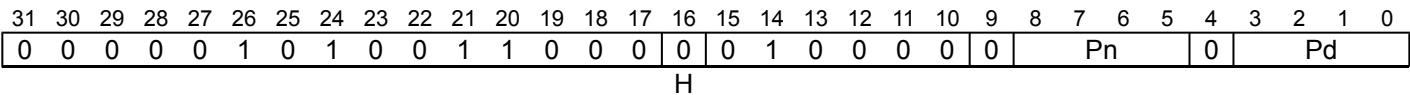
## High half



PUNPKHI <Pd>.H, <Pn>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean hi = TRUE;
```

## Low half



PUNPKLO <Pd>.H, <Pn>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean hi = FALSE;
```

## Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) operand = P[n, PL];
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit pbit = PredicateElement(operand, if hi then e + elements else e, esize DIV 2);
    Elem[result, e, psize] = ZeroExtend(pbit, psize);

P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:



- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

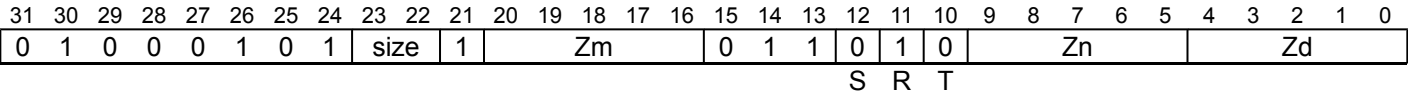
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RADDHNB

Rounding add narrow high part (bottom)

Add each vector element of the first source vector to the corresponding vector element of the second source vector, and place the most significant rounded half of the result in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. This instruction is unpredicated.



```
RADDHNB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = ((element1 + element2) + (1 << (halfesize - 1))) >> halfesize;
    Elem[result, 2*e + 0, halfesize] = res<halfesize-1:0>;
    Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

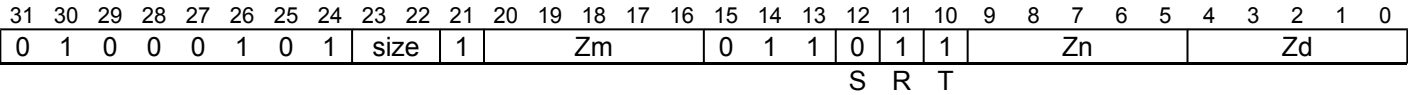
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RADDHNT

Rounding add narrow high part (top)

Add each vector element of the first source vector to the corresponding vector element of the second source vector, and place the most significant rounded half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.



```
RADDHNT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = ((element1 + element2) + (1 << (halfesize - 1))) >> halfesize;
    Elem[result, 2*e + 1, halfesize] = res<halfesize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RAX1

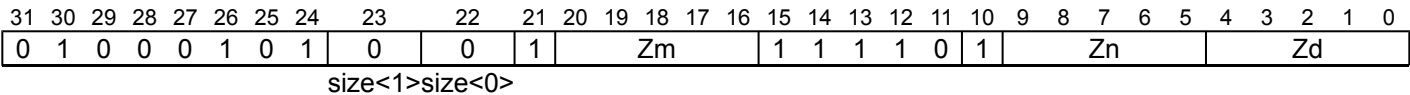
Bitwise rotate left by 1 and exclusive OR

Rotate each 64-bit element of the second source vector left by 1 and exclusive OR with the corresponding elements of the first source vector. The results are placed in the corresponding elements of the destination vector. This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.SHA3 indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled, or FEAT\_SME2p1 is implemented.

SVE2
(FEAT\_SVE\_SHA3)



RAX1 <Zd>.D, <Zn>.D, <Zm>.D

```
if !HaveSVE2SHA3() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
if HaveSME2p1() then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 64;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(64) element1 = Elem[operand1, e, 64];
    bits(64) element2 = Elem[operand2, e, 64];
    Elem[result, e, 64] = element1 EOR ROL(element2, 1);
Z[d, VL] = result;
```

Operational information

- If PSTATE.DIT is 1:
  - The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

RBIT

Reverse bits (predicated)

Reverse bits in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	0	0	1	1	1	1	0	0	Pg													

```
RBIT <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = BitReverse(element);

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



RDFFR (predicated)

Return predicate of succesfully loaded elements

Read the first-fault register (FFR) and place active elements in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags. This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	0	0	1	1	0	0	0	1	1	1	1	0	0	0			Pg		0			Pd	
S																															

RDFFR <Pd>.B, <Pg>/Z

```
if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(PL) ffr = FFR[PL];
bits(PL) result = ffr AND mask;

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, 8);
P[d, PL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

RDFFR (unpredicated)

Read the first-fault register

Read the first-fault register (FFR) and place in the destination predicate without predication.  
This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	0	0	1	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0				Pd
										S																					

**RDFFR** <Pd>.B

```
if !HaveSVE() then UNDEFINED;
integer d = UInt(Pd);
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) ffr = FFR[PL];
P[d, PL] = ffr;
```

Operational information

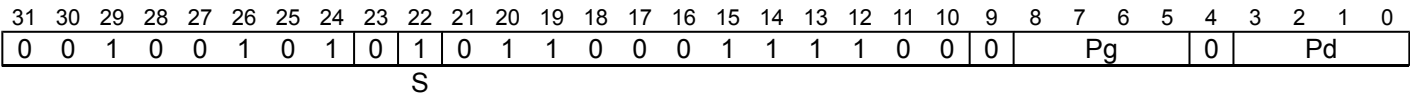
- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

RDFFRS

Return predicate of succesfully loaded elements, setting the condition flags

Read the first-fault register (FFR) and place active elements in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.



RDFFRS <Pd>.B, <Pg>/Z

```
if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(PL) ffr = FFR[PL];
bits(PL) result = ffr AND mask;

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, 8);
P[d, PL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

RDVL

Read multiple of vector register size to scalar register

Multiply the current vector register size in bytes by an immediate in the range -32 to 31 and place the result in the 64-bit destination general-purpose register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	0	1	0	1	0	imm6						Rd				

```
RDVL <Xd>, #<imm>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer d = UInt(Rd);
integer imm = SInt(imm6);
```

Assembler Symbols

- <Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.
- <imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
integer len = imm * (VL DIV 8);
X[d, 64] = len<63:0>;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

REV (predicate)

Reverse all elements in a predicate

Reverse the order of all elements in the source predicate and place in the destination predicate. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	1	0	1	size	1	1	0	1	0	0	0	1	0	0	0	0	0	0	0	Pn				0	Pd			

REV <Pd>.<T>, <Pn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer d = UInt(Pd);
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) operand = P[n, PL];
bits(PL) result = Reverse(operand, esize DIV 8);
P[d, PL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

REV (vector)

Reverse all elements in a vector (unpredicated)

Reverse the order of all elements in the source vector and place in the destination vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	1	1	0	0	0	0	0	1	1	1	0	Zn				Zd						

```
REV <Zd>.<T>, <Zn>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(VL) operand = Z[n, VL];
bits(VL) result = Reverse(operand, esize);
Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

REVB, REVH, REVW

Reverse bytes / halfwords / words within elements (predicated)

Reverse the order of 8-bit bytes, 16-bit halfwords or 32-bit words within each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

It has encodings from 3 classes: [Byte](#) , [Halfword](#) and [Word](#)

Byte

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	0	0	1	0	0	1	0	0	Pg	Zn				Zd								

REVB <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer swsize = 8;
```

Halfword

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	0	0	1	0	1	1	0	0	Pg			Zn					Zd					

REVH <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer swsize = 16;
```

Word

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size		1	0	0	1	1	0	1	0	0	Pg			Zn				Zd					

REWV <Zd>.D, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size != '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer swsize = 32;
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <T> For the byte variant: is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

For the halfword variant: is the size specifier, encoded in "size<0>":

size<0>	<T>
0	S
1	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

## Operation

```

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = Reverse(element, swsize);

Z[d, VL] = result;

```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

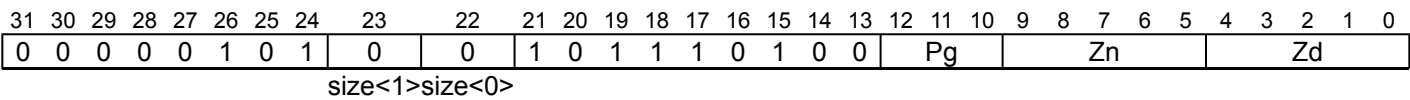


# REVD

Reverse 64-bit doublewords in elements (predicated)

Reverse the order of 64-bit doublewords within each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

## SVE2 (FEAT\_SVE2p1)



REVD <Zd>.Q, <Pg>/M, <Zn>.Q

```
if !HaveSME() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 128;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer swsize = 64;
```

### Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = Reverse(element, swsize);

Z[d, VL] = result;
```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

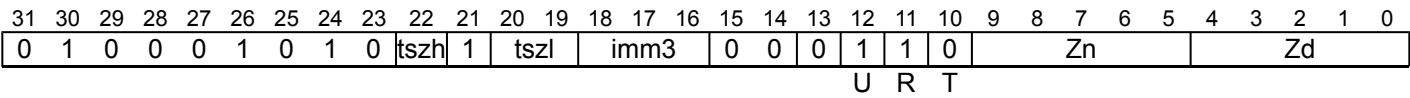
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RSHRNB

Rounding shift right narrow by immediate (bottom)

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



```
RSHRNB <Zd>.<T>, <Zn>.<Tb>, #<const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = (UInt(element) + (1 << (shift-1))) >> shift;
    Elem[result, 2*e + 0, esize] = res<esize-1:0>;
    Elem[result, 2*e + 1, esize] = Zeros(esize);

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

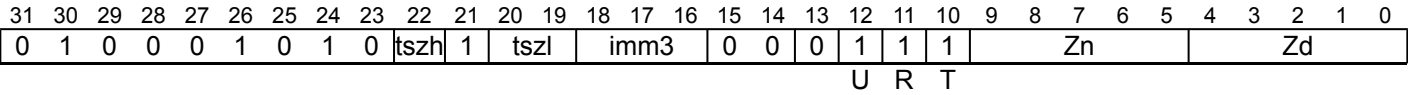
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RSHRNT

Rounding shift right narrow by immediate (top)

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



```
RSHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = (UInt(element) + (1 << (shift-1))) >> shift;
    Elem[result, 2*e + 1, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

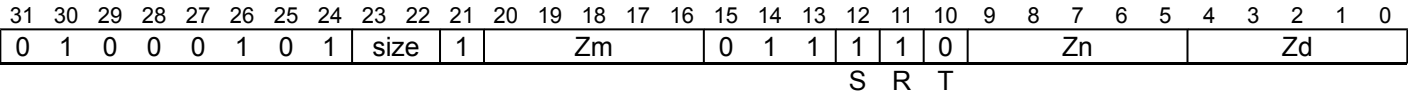
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RSUBHNB

Rounding subtract narrow high part (bottom)

Subtract each vector element of the second source vector from the corresponding vector element in the first source vector, and place the most significant rounded half of the result in the even-numbered half-width destination elements, while setting the odd-numbered half-width destination elements to zero. This instruction is unpredicated.



```
RSUBHNB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = ((element1 - element2) + (1 << (halfesize - 1))) >> halfesize;
    Elem[result, 2*e + 0, halfesize] = res<halfesize-1:0>;
    Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

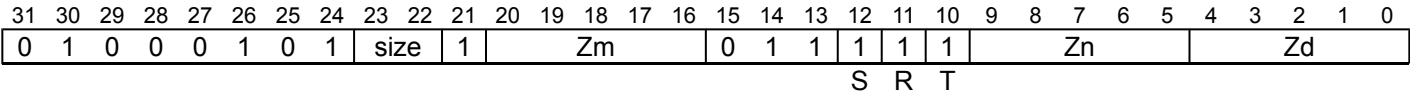
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



RSUBHNT

Rounding subtract narrow high part (top)

Subtract each vector element of the second source vector from the corresponding vector element in the first source vector, and place the most significant rounded half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.



```
RSUBHNT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = ((element1 - element2) + (1 << (halfesize - 1))) >> halfesize;
    Elem[result, 2*e + 1, halfesize] = res<halfesize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

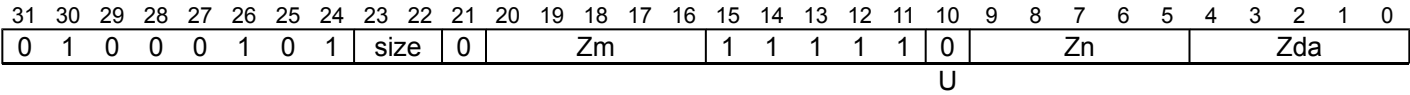
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SABA

Signed absolute difference and accumulate

Compute the absolute difference between signed integer values in elements of the second source vector and corresponding elements of the first source vector, and add the difference to the corresponding elements of the destination vector. This instruction is unpredicated.



```
SABA <Zda>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;

Z[da, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

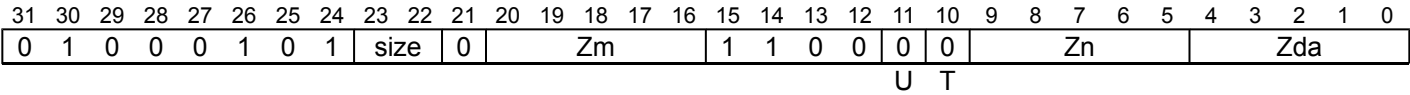
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SABALB

Signed absolute difference and accumulate long (bottom)

Compute the absolute difference between even-numbered signed integer values in elements of the second source vector and corresponding elements of the first source vector, and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.



SABALB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;

Z[da, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

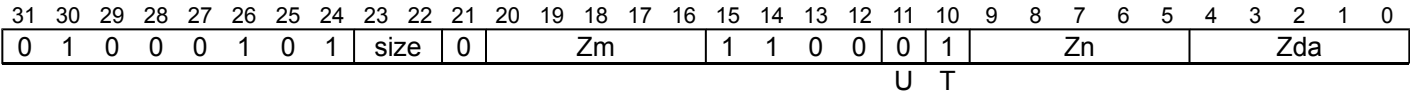
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SABALT

Signed absolute difference and accumulate long (top)

Compute the absolute difference between odd-numbered signed elements of the second source vector and corresponding elements of the first source vector, and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.



SABALT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;

Z[da, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

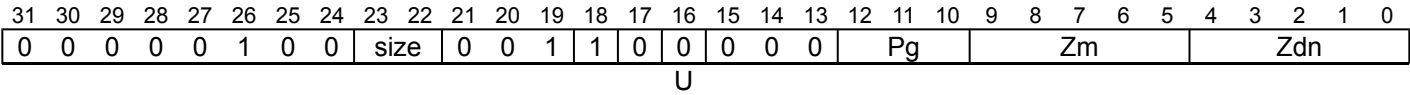
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SABD

Signed absolute difference (predicated)

Compute the absolute difference between signed integer values in active elements of the second source vector and corresponding elements of the first source vector and destructively place the difference in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



```
SABD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if ActivePredicateElement(mask, e, esize) then
        integer absdiff = Abs(element1 - element2);
        Elem[result, e, esize] = absdiff<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

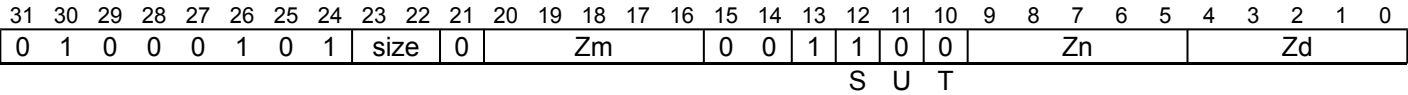
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SABDLB

Signed absolute difference long (bottom)

Compute the absolute difference between even-numbered signed integer values in elements of the second source vector and corresponding elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



SABDLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    integer res = Abs(element1 - element2);
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

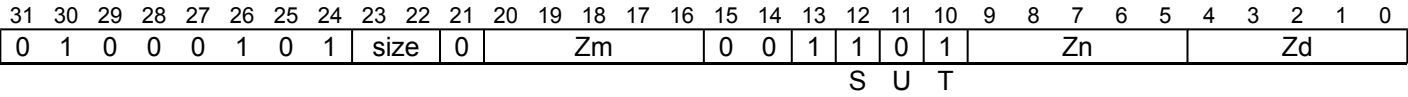
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SABDLT

Signed absolute difference long (top)

Compute the absolute difference between odd-numbered signed integer values in elements of the second source vector and corresponding elements of the first source vector, and place the results in overlapping double-width elements of the destination vector. This instruction is unpredicated.



SABDLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    integer res = Abs(element1 - element2);
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

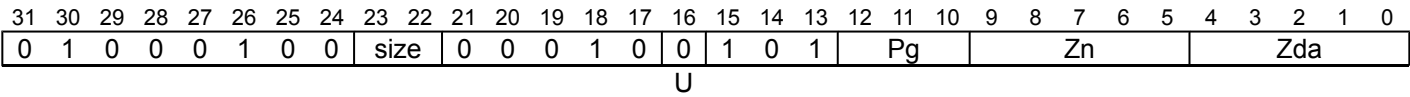
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SADALP

Signed add and accumulate long pairwise

Add pairs of adjacent signed integer values and accumulate the results into the overlapping double-width elements of the destination vector.



SADALP <Zda>.<T>, <Pg>/M, <Zn>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the second source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand_acc = Z[da, VL];
bits(VL) operand_src = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if !ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Elem[operand_acc, e, esize];
    else
        integer element1 = SInt(Elem[operand_src, 2*e + 0, esize DIV 2]);
        integer element2 = SInt(Elem[operand_src, 2*e + 1, esize DIV 2]);
        bits(esize) sum = (element1 + element2)<esize-1:0>;
        Elem[result, e, esize] = Elem[operand_acc, e, esize] + sum;

Z[da, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

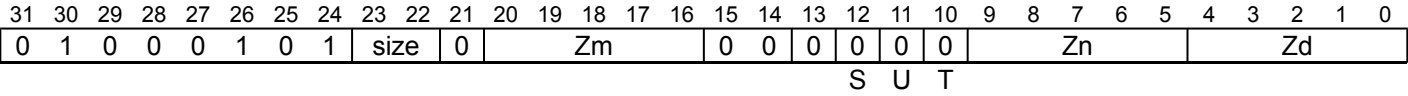
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SADDLB

Signed add long (bottom)

Add the corresponding even-numbered signed elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



```
SADDLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 0;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 + element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

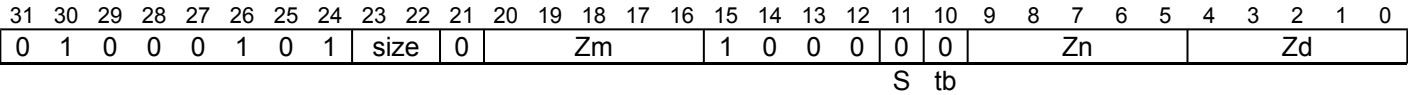
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SADDLBT

Signed add long (bottom + top)

Add the even-numbered signed elements of the first source vector to the odd-numbered signed elements of the second source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



```
SADDLBT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 + element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

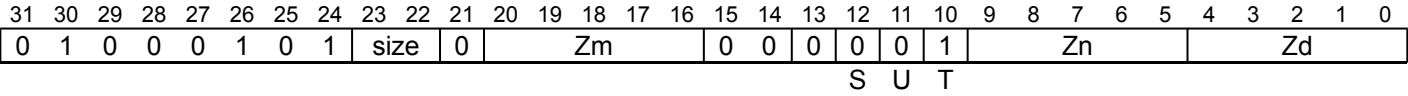
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SADDLT

Signed add long (top)

Add the corresponding odd-numbered signed elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



SADDLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 + element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

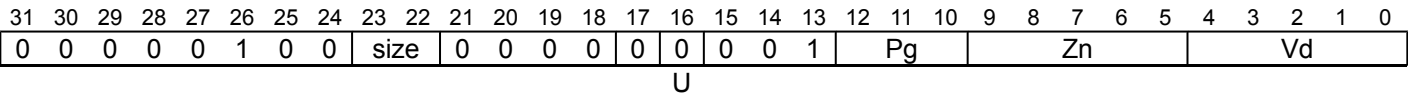
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SADDV

Signed add reduction to scalar

Signed add horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Narrow elements are first sign-extended to 64 bits. Inactive elements in the source vector are treated as zero.



SADDV <Dd>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

- <Dd> Is the 64-bit name of the destination SIMD&FP register, encoded in the "Vd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	RESERVED

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = Z[n, VL];
integer sum = 0;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer element = SInt(Elem[operand, e, esize]);
        sum = sum + element;

V[d, 64] = sum<63:0>;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

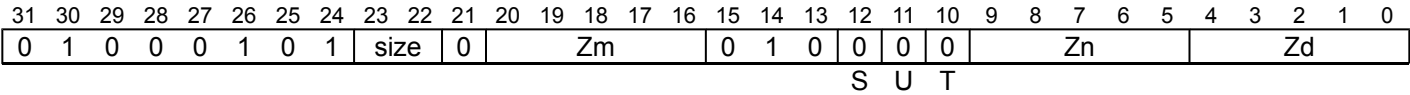




SADDWB

Signed add wide (bottom)

Add the even-numbered signed elements of the second source vector to the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.



```
SADDWB <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    Elem[result, e, esize] = (element1 + element2)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

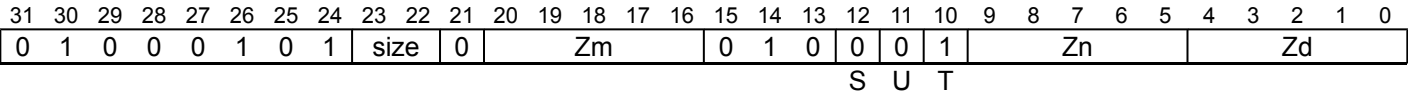
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SADDWT

Signed add wide (top)

Add the odd-numbered signed elements of the second source vector to the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.



```
SADDWT <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    Elem[result, e, esize] = (element1 + element2)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

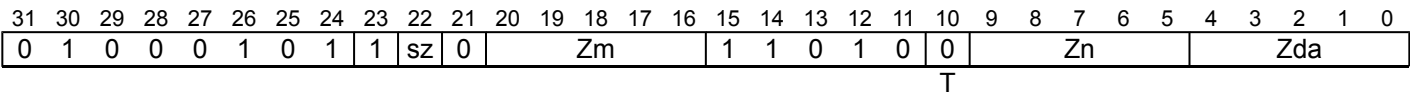
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SBCLB

Subtract with carry long (bottom)

Subtract the even-numbered elements of the first source vector and the inverted 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector from the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.



```
SBCLB <Zda>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand = Z[n, VL];
bits(VL) carries = Z[m, VL];
bits(VL) result = Z[da, VL];

for p = 0 to pairs-1
    bits(esize) element1 = Elem[result, 2*p + 0, esize];
    bits(esize) element2 = Elem[operand, 2*p + 0, esize];
    bit carry_in = Elem[carries, 2*p + 1, esize]<0>;

    (res, nzcv) = AddWithCarry(element1, NOT(element2), carry_in);
    carry_out = nzcv<1>;

    Elem[result, 2*p + 0, esize] = res;
    Elem[result, 2*p + 1, esize] = ZeroExtend(carry_out, esize);

Z[da, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

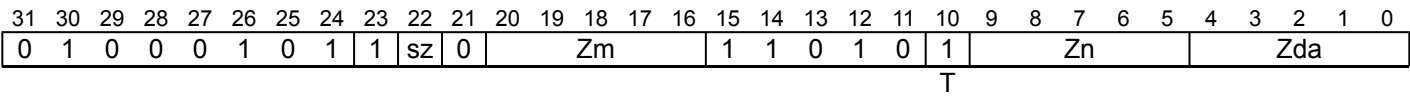
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SBCLT

Subtract with carry long (top)

Subtract the odd-numbered elements of the first source vector and the inverted 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector from the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.



```
SBCLT <Zda>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand = Z[n, VL];
bits(VL) carries = Z[m, VL];
bits(VL) result = Z[da, VL];

for p = 0 to pairs-1
    bits(esize) element1 = Elem[result, 2*p + 0, esize];
    bits(esize) element2 = Elem[operand, 2*p + 1, esize];
    bit carry_in = Elem[carries, 2*p + 1, esize]<0>;

    (res, nzcv) = AddWithCarry(element1, NOT(element2), carry_in);
    carry_out = nzcv<1>;

    Elem[result, 2*p + 0, esize] = res;
    Elem[result, 2*p + 1, esize] = ZeroExtend(carry_out, esize);

Z[da, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

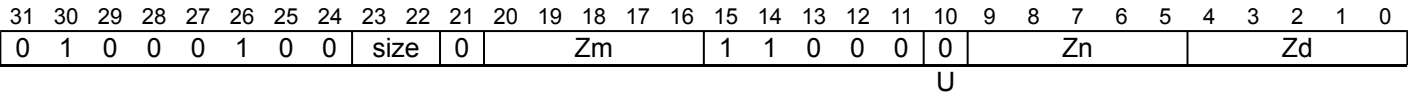


SCLAMP

Signed clamp to minimum/maximum vector

Clamp each signed element in the destination vector to between the signed minimum value in the corresponding element of the first source vector and the signed maximum value in the corresponding element of the second source vector and destructively write the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE2
(FEAT\_SVE2p1)



SCLAMP <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSME() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[d, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    integer element3 = SInt(Elem[operand3, e, esize]);
    integer res = Min(Max(element1, element3), element2);
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SCVTF

Signed integer convert to floating-point (predicated)

Convert to floating-point from the signed integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.

It has encodings from 7 classes: [16-bit to half-precision](#) , [32-bit to half-precision](#) , [32-bit to single-precision](#) , [32-bit to double-precision](#) , [64-bit to half-precision](#) , [64-bit to single-precision](#) and [64-bit to double-precision](#)

16-bit to half-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	1	Pg			Zn				Zd					

int\_U

SCVTF <Zd>.H, <Pg>/M, <Zn>.H

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 16;
constant integer d_esign = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

32-bit to half-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	0	1	0	1	0	1	0	0	1	0	1	Pg			Zn				Zd					

int\_U

SCVTF <Zd>.H, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 32;
constant integer d_esign = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

32-bit to single-precision

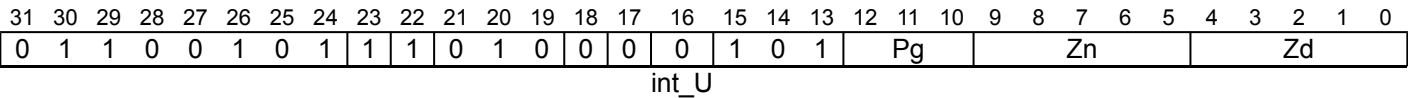
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	1	0	0	1	0	1	0	0	1	0	1	Pg			Zn				Zd					

int\_U

SCVTF <Zd>.S, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 32;
constant integer d_esign = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

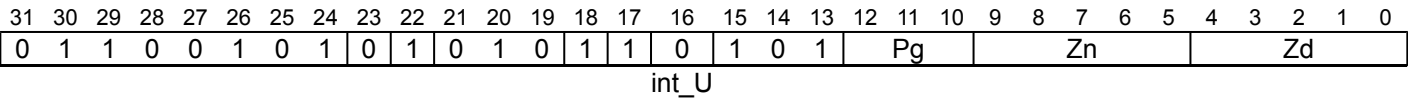
32-bit to double-precision



SCVTF <Zd>.D, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 32;
constant integer d_esign = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

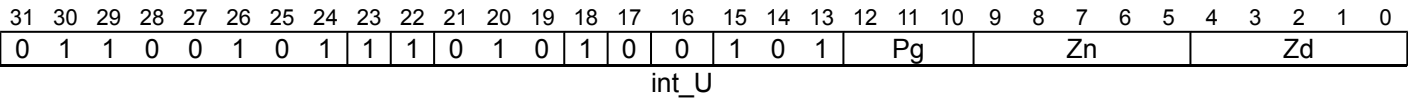
64-bit to half-precision



SCVTF <Zd>.H, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 64;
constant integer d_esign = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

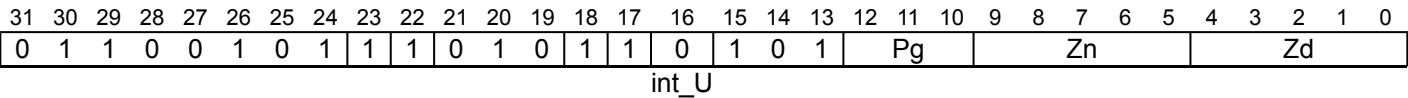
64-bit to single-precision



SCVTF <Zd>.S, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 64;
constant integer d_esign = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

64-bit to double-precision



SCVTF <Zd>.D, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 64;
constant integer d_esign = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        bits(d_esign) fpval = FixedToFP(element<s_esign-1:0>, 0, unsigned, FPCR[], rounding, d_esign);
        Elem[result, e, esize] = ZeroExtend(fpval, esize);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

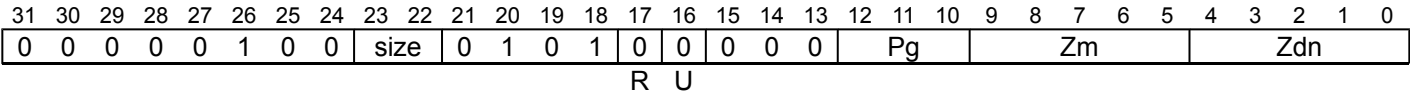
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SDIV

Signed divide (predicated)

Signed divide active elements of the first source vector by corresponding elements of the second source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



```
SDIV <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size<0>":

size<0>	<T>
0	S
1	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if ActivePredicateElement(mask, e, esize) then
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer quotient;
        if element2 == 0 then
            quotient = 0;
        else
            quotient = RoundTowardsZero(Real(element1) / Real(element2));
        Elem[result, e, esize] = quotient<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

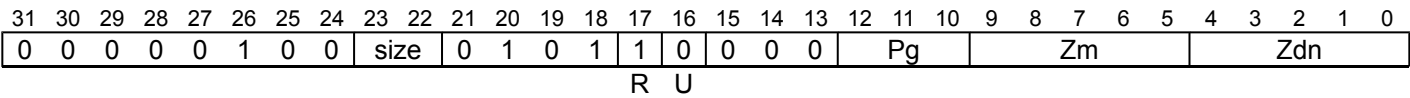
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SDIVR

Signed reversed divide (predicated)

Signed reversed divide active elements of the second source vector by corresponding elements of the first source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



```
SDIVR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size<0>":

size<0>	<T>
0	S
1	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if ActivePredicateElement(mask, e, esize) then
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer quotient;
        if element1 == 0 then
            quotient = 0;
        else
            quotient = RoundTowardsZero(Real(element2) / Real(element1));
        Elem[result, e, esize] = quotient<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

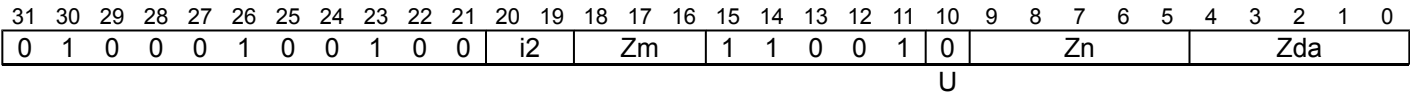
## SDOT (2-way, indexed)

Signed integer indexed dot product

The signed integer indexed dot product instruction computes the dot product of a group of two signed 16-bit integer values held in each 32-bit element of the first source vector multiplied by a group of two signed 16-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated.

### SVE2 (FEAT\_SVE2p1)



SDOT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index of a group of two 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 1
        integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
        integer element2 = SInt(Elem[operand2, 2 * s + i, esize DIV 2]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

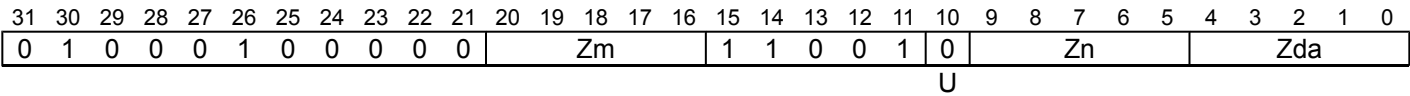
## SDOT (2-way, vectors)

Signed integer dot product

The signed integer dot product instruction computes the dot product of a group of two signed 16-bit integer values held in each 32-bit element of the first source vector multiplied by a group of two signed 16-bit integer values in the corresponding 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

This instruction is unpredicated.

### SVE2 (FEAT\_SVE2p1)



SDOT <Zda>.S, <Zn>.H, <Zm>.H

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
  bits(esize) res = Elem[operand3, e, esize];
  for i = 0 to 1
    integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2 * e + i, esize DIV 2]);
    res = res + element1 * element2;
  Elem[result, e, esize] = res;

Z[da, VL] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



## SDOT (4-way, indexed)

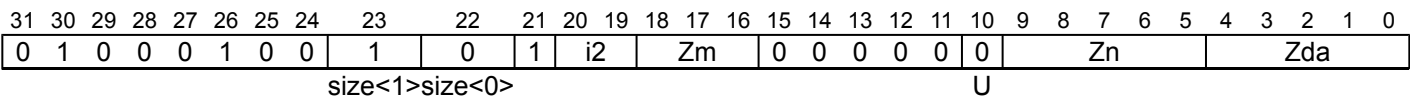
Signed integer indexed dot product

The signed integer indexed dot product instruction computes the dot product of a group of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four signed 8-bit or 16-bit integer values in an indexed 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group. This instruction is unpredicated.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

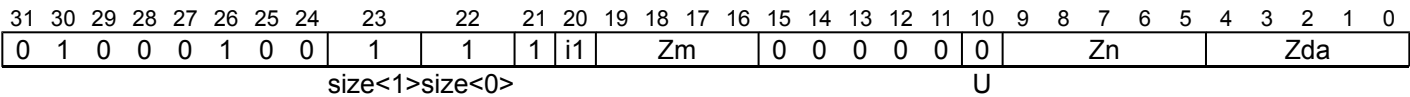
### 32-bit



SDOT <Zda>.S, <Zn>.B, <Zm>.B[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### 64-bit



SDOT <Zda>.D, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.  
For the 64-bit variant: is the immediate index of a 64-bit group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the "i1" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

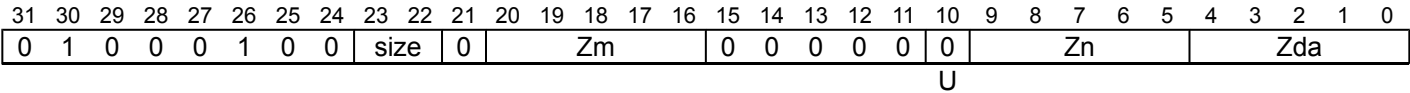


SDOT (4-way, vectors)

Signed integer dot product

The signed integer dot product instruction computes the dot product of a group of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four signed 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector.

This instruction is unpredicated.



SDOT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size<0>":

size<0>	<T>
0	S
1	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size<0>":

size<0>	<Tb>
0	B
1	H

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
  bits(esize) res = Elem[operand3, e, esize];
  for i = 0 to 3
    integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
    integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
    res = res + element1 * element2;
  Elem[result, e, esize] = res;

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SEL (predicates)

Conditionally select elements from two predicates

Read active elements from the first source predicate and inactive elements from the second source predicate and place in the corresponding elements of the destination predicate. Does not set the condition flags.

This instruction is used by the alias [MOV \(predicate, predicated, merging\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	0	0	0	Pm				0	1	Pg				1	Pn				1	Pd			
S																															

SEL <Pd>.B, <Pg>, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
<a href="#">MOV (predicate, predicated, merging)</a>	Pd == Pm

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit element1 = PredicateElement(operand1, e, esize);
    bit element2 = PredicateElement(operand2, e, esize);
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, psize] = ZeroExtend(element1, psize);
    else
        Elem[result, e, psize] = ZeroExtend(element2, psize);

P[d, PL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SEL (vectors)

Conditionally select elements from two vectors

Select elements from the first source vector where the corresponding vector select predicate element is true, and from the second source vector where the predicate element is false, placing them in the corresponding elements of the destination vector.

This instruction is used by the alias [MOV \(vector, predicated\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	Zm				1	1	Pv			Zn				Zd								

SEL <Zd>.<T>, <Pv>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer v = UInt(Pv);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pv> Is the name of the vector select predicate register, encoded in the "Pv" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Alias Conditions

Alias	Is preferred when
<a href="#">MOV (vector, predicated)</a>	Zd == Zm

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[v, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(NOT(mask), esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Elem[operand1, e, esize];
    else
        Elem[result, e, esize] = Elem[operand2, e, esize];

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SETFFR

Initialise the first-fault register to all true

Initialise the first-fault register (FFR) to all true prior to a sequence of first-fault or non-fault loads. This instruction is unpredicated.  
This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	0	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

SETFFR

```
if !HaveSVE() then UNDEFINED;
```

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
FFR[PL] = Ones(PL);
```

Operational information

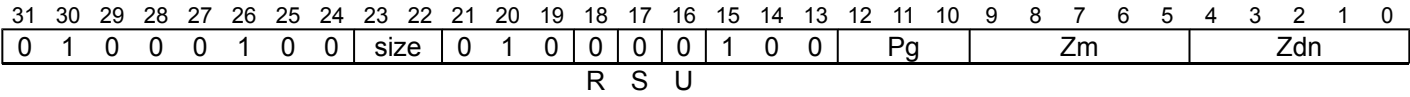
If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

SHADD

Signed halving addition

Add active signed elements of the first source vector to corresponding signed elements of the second source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



```
SHADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = (element1 + element2) >> 1;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.



- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

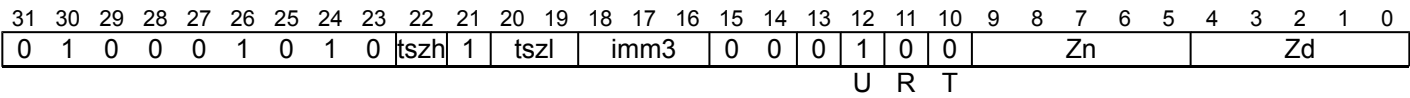
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHRNB

Shift right narrow by immediate (bottom)

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



```
SHRNB <Zd>.<T>, <Zn>.<Tb>, #<const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = UInt(element) >> shift;
    Elem[result, 2*e + 0, esize] = res<esize-1:0>;
    Elem[result, 2*e + 1, esize] = Zeros(esize);

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

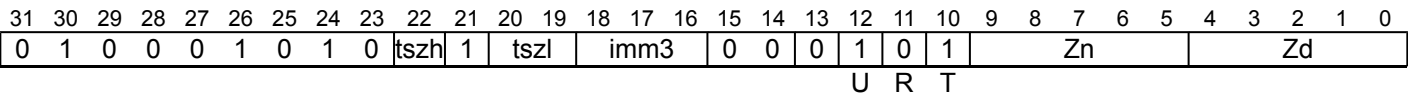
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHRNT

Shift right narrow by immediate (top)

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



```
SHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = UInt(element) >> shift;
    Elem[result, 2*e + 1, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

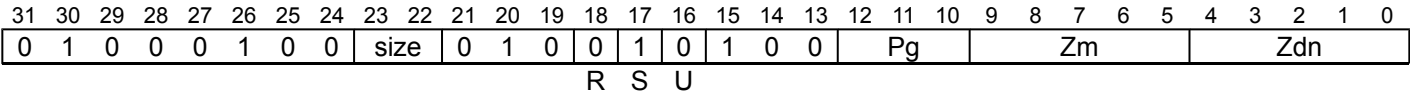
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHSUB

Signed halving subtract

Subtract active signed elements of the second source vector from corresponding signed elements of the first source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



```
SHSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = (element1 - element2) >> 1;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

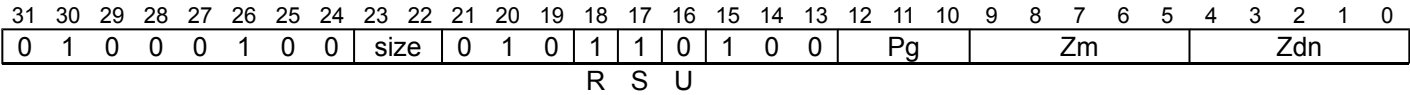
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHSUBR

Signed halving subtract reversed vectors

Subtract active signed elements of the first source vector from corresponding signed elements of the second source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



```
SHSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = (element2 - element1) >> 1;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.



- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SLI

Shift left and insert (immediate)

Shift each source vector element left by an immediate value, and insert the result into the corresponding vector element in the destination vector register, merging the shifted bits from each source element with existing bits in each destination vector element. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	tszh	0	tszl	imm3			1	1	1	1	0	1	Zn						Zd					

SLI <Zd>.<T>, <Zn>.<T>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(esize) element1 = Elem[result, e, esize];
    bits(esize) element2 = Elem[operand, e, esize];
    bits(esize) mask = LSL(Ones(esize), shift);
    bits(esize) shiftedval = LSL(element2, shift);
    Elem[result, e, esize] = (element1 AND (NOT mask)) OR shiftedval;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SM4E

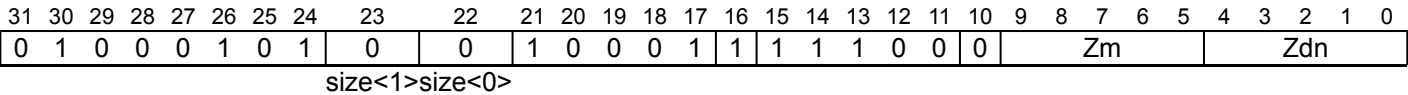
SM4 encryption and decryption

The SM4E instruction reads 16 bytes of input data from each 128-bit segment of the first source vector, together with four iterations of 32-bit round keys from the corresponding 128-bit segments of the second source vector. Each block of data is encrypted by four rounds in accordance with the SM4 standard, and destructively placed in the corresponding segments of the first source vector. This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.SM4 indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE2  
(FEAT\_SVE\_SM4)



SM4E <Zdn>.S, <Zdn>.S, <Zm>.S

```
if !HaveSVE() || !HaveSVE2SM4() then UNDEFINED;
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for s = 0 to segments-1
    bits(128) key = Elem[operand2, s, 128];
    bits(32) intval;
    bits(8) sbboxout;
    bits(128) roundresult = Elem[operand1, s, 128];
    bits(32) roundkey;

    for index = 0 to 3
        roundkey = Elem[key, index, 32];
        intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;

        for i = 0 to 3
            Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);

        intval = intval EOR ROL(intval, 2) EOR ROL(intval, 10) EOR ROL(intval, 18) EOR ROL(intval, 24);
        intval = intval EOR roundresult<31:0>;

        roundresult<31:0> = roundresult<63:32>;
        roundresult<63:32> = roundresult<95:64>;
        roundresult<95:64> = roundresult<127:96>;
        roundresult<127:96> = intval;

    Elem[result, s, 128] = roundresult;

Z[dn, VL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SM4EKEY

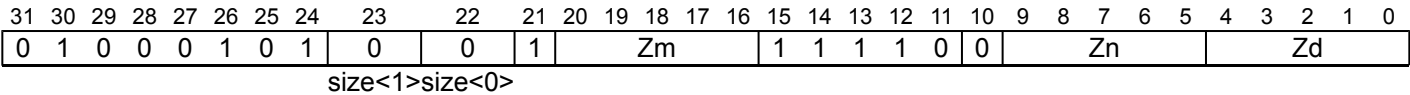
SM4 key updates

The `SM4EKEY` instruction reads four rounds of 32-bit input key values from each 128-bit segment of the first source vector, along with four rounds of 32-bit constants from the corresponding 128-bit segment of the second source vector. The four rounds of output key values are derived in accordance with the SM4 standard, and placed in the corresponding segments of the destination vector. This instruction is unpredicated.

`ID_AA64ZFR0_EL1.SM4` indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless `FEAT_SME_FA64` is implemented and enabled.

SVE2  
(FEAT\_SVE\_SM4)



`SM4EKEY <Zd>.S, <Zn>.S, <Zm>.S`

```
if !HaveSVE() || !HaveSVE2SM4() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for s = 0 to segments-1
    bits(128) source = Elem[operand2, s, 128];
    bits(32) intval;
    bits(8) sbxout;
    bits(32) const;
    bits(128) roundresult = Elem[operand1, s, 128];

    for index = 0 to 3
        const = Elem[source, index, 32];
        intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;
        for i = 0 to 3
            Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);

        intval = intval EOR ROL(intval, 13) EOR ROL(intval, 23);
        intval = intval EOR roundresult<31:0>;

        roundresult<31:0> = roundresult<63:32>;
        roundresult<63:32> = roundresult<95:64>;
        roundresult<95:64> = roundresult<127:96>;
        roundresult<127:96> = intval;

    Elem[result, s, 128] = roundresult;

Z[d, VL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

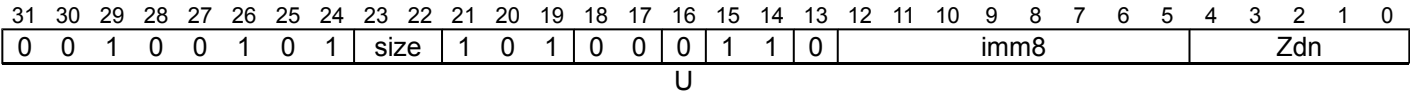
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMAX (immediate)

Signed maximum with immediate (unpredicated)

Determine the signed maximum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.



```
SMAX <Zdn>.<T>, <Zdn>.<T>, #<imm>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;
integer imm = Int(imm8, unsigned);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm> Is the signed immediate operand, in the range -128 to 127, encoded in the "imm8" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    Elem[result, e, esize] = Max(element1, imm)<esize-1:0>;

Z[dn, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.



- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

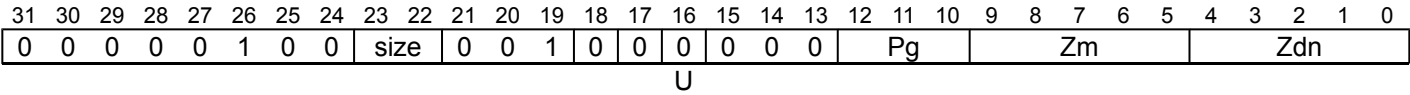
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMAX (vectors)

Signed maximum vectors (predicated)

Determine the signed maximum of active elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



```
SMAX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if ActivePredicateElement(mask, e, esize) then
        integer maximum = Max(element1, element2);
        Elem[result, e, esize] = maximum<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

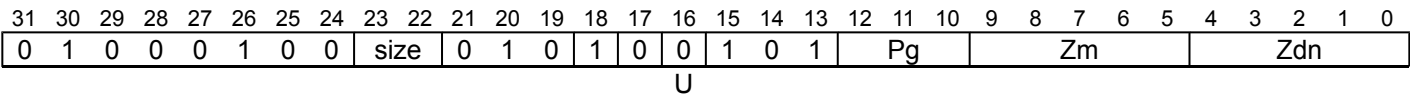
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMAXP

Signed maximum pairwise

Compute the maximum value of each pair of adjacent signed integer elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.



```
SMAXP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
integer element1;
integer element2;

for e = 0 to elements-1
  if !ActivePredicateElement(mask, e, esize) then
    Elem[result, e, esize] = Elem[operand1, e, esize];
  else
    if IsEven(e) then
      element1 = SInt(Elem[operand1, e + 0, esize]);
      element2 = SInt(Elem[operand1, e + 1, esize]);
    else
      element1 = SInt(Elem[operand2, e - 1, esize]);
      element2 = SInt(Elem[operand2, e + 0, esize]);
    integer res = Max(element1, element2);
    Elem[result, e, esize] = res<esize-1:0>;

Z[dn, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

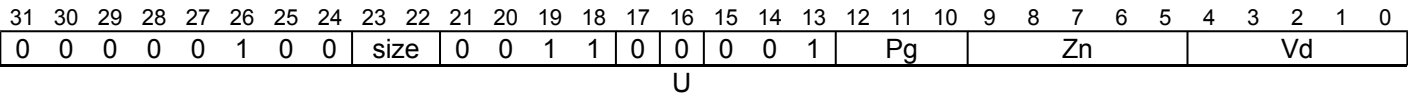
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMAXQV

Signed maximum reduction of quadword vector segments

Signed maximum of the same element numbers from each 128-bit source vector segment, placing each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as the minimum signed integer for the element size.

SVE2
(FEAT\_SVE2p1)



SMAXQV <Vd>.<T>, <Pg>, <Zn>.<Tb>

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = FALSE;
```

Assembler Symbols

<Vd> Is the name of the destination SIMD&FP register, encoded in the "Vd" field.

<T> Is an arrangement specifier, encoded in "size":

size	<T>
00	16B
01	8H
10	4S
11	2D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	B
01	H
10	S
11	D

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elemperssegment = 128 DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(128) result = Zeros(128);
bits(128) stmp = Zeros(128);

integer dtmp;

for e = 0 to elemperssegment-1
    dtmp = if unsigned then 0 else -(2^(esize-1));
    for s = 0 to segments-1
        if ActivePredicateElement(mask, s * elemperssegment + e, esize) then
            stmp = Elem[operand, s, 128];
            dtmp = Max(dtmp, SInt(Elem[stmp, e, esize]));
        Elem[result, e, esize] = dtmp<esize-1:0>;

V[d, 128] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

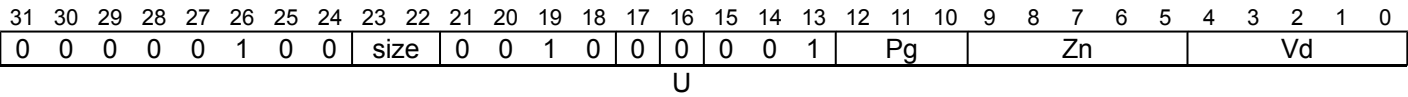
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMA XV

Signed maximum reduction to scalar

Signed maximum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the minimum signed integer for the element size.



SMA XV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = FALSE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
integer maximum = if unsigned then 0 else -(2^(esize-1));

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer element = Int(Elem[operand, e, esize], unsigned);
        maximum = Max(maximum, element);

V[d, esize] = maximum<esize-1:0>;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:



- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

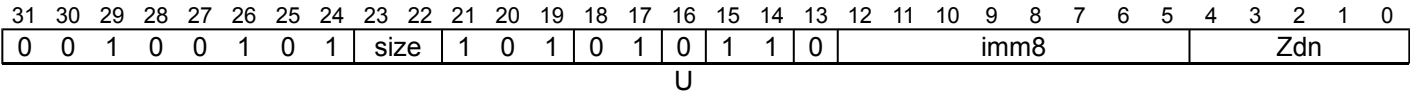
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMIN (immediate)

Signed minimum with immediate (unpredicated)

Determine the signed minimum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.



```
SMIN <Zdn>.<T>, <Zdn>.<T>, #<imm>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;
integer imm = Int(imm8, unsigned);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm> Is the signed immediate operand, in the range -128 to 127, encoded in the "imm8" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    Elem[result, e, esize] = Min(element1, imm)<esize-1:0>;

Z[dn, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

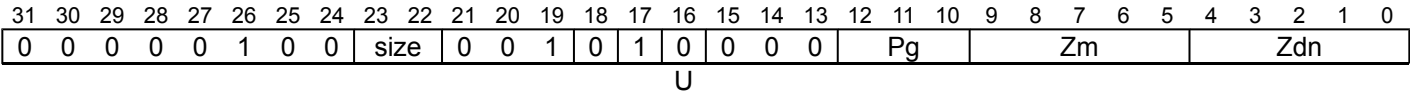
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMIN (vectors)

Signed minimum vectors (predicated)

Determine the signed minimum of active elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



```
SMIN <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if ActivePredicateElement(mask, e, esize) then
        integer minimum = Min(element1, element2);
        Elem[result, e, esize] = minimum<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

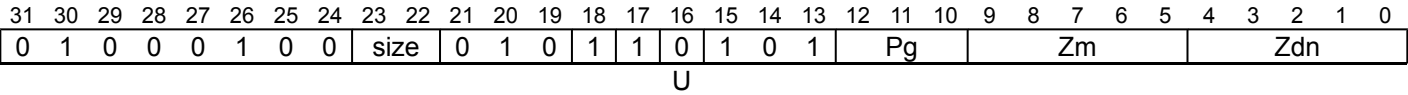
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMINP

Signed minimum pairwise

Compute the minimum value of each pair of adjacent signed integer elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.



```
SMINP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
integer element1;
integer element2;

for e = 0 to elements-1
  if !ActivePredicateElement(mask, e, esize) then
    Elem[result, e, esize] = Elem[operand1, e, esize];
  else
    if IsEven(e) then
      element1 = SInt(Elem[operand1, e + 0, esize]);
      element2 = SInt(Elem[operand1, e + 1, esize]);
    else
      element1 = SInt(Elem[operand2, e - 1, esize]);
      element2 = SInt(Elem[operand2, e + 0, esize]);
    integer res = Min(element1, element2);
    Elem[result, e, esize] = res<esize-1:0>;

Z[dn, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

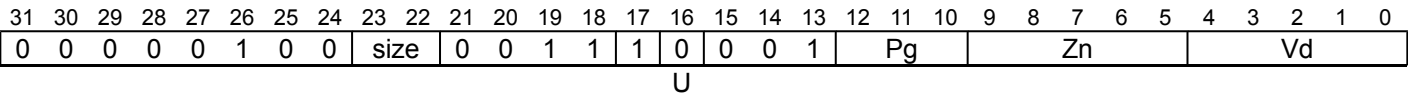
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMINQV

Signed minimum reduction of quadword vector segments

Signed minimum of the same element numbers from each 128-bit source vector segment, placing each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as the maximum signed integer for the element size.

SVE2  
(FEAT\_SVE2p1)



SMINQV <Vd>.<T>, <Pg>, <Zn>.<Tb>

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = FALSE;
```

Assembler Symbols

<Vd> Is the name of the destination SIMD&FP register, encoded in the "Vd" field.

<T> Is an arrangement specifier, encoded in "size":

size	<T>
00	16B
01	8H
10	4S
11	2D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	B
01	H
10	S
11	D



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elempersegment = 128 DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(128) result = Zeros(128);
bits(128) stmp = Zeros(128);

integer dtmp;

for e = 0 to elempersegment-1
    dtmp = if unsigned then (2^esize - 1) else (2^(esize-1) - 1);
    for s = 0 to segments-1
        if ActivePredicateElement(mask, s * elempersegment + e, esize) then
            stmp = Elem[operand, s, 128];
            dtmp = Min(dtmp, SInt(Elem[stmp, e, esize]));
        Elem[result, e, esize] = dtmp<esize-1:0>;

V[d, 128] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

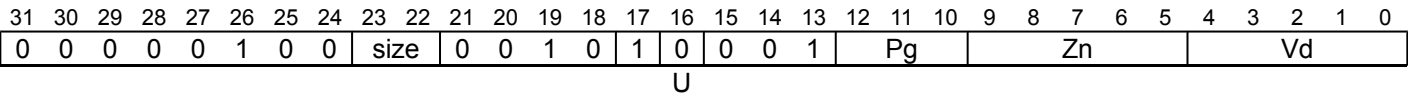
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMINV

Signed minimum reduction to scalar

Signed minimum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the maximum signed integer for the element size.



SMINV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = FALSE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
integer minimum = if unsigned then (2^esize - 1) else (2^(esize-1) - 1);

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer element = Int(Elem[operand, e, esize], unsigned);
        minimum = Min(minimum, element);

V[d, esize] = minimum<esize-1:0>;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SMLALB (indexed)

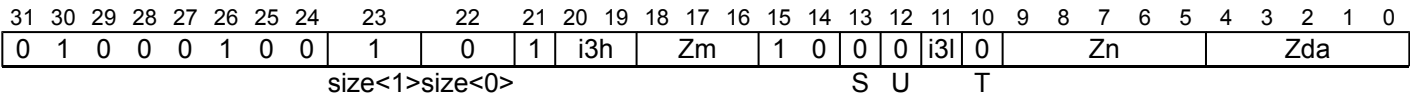
Signed multiply-add long to accumulator (bottom, indexed)

Multiply the even-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment and destructively add to the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

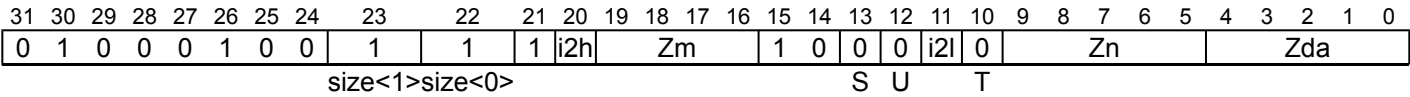
### 32-bit



**SMLALB** <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

### 64-bit



**SMLALB** <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + product;

Z[da, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

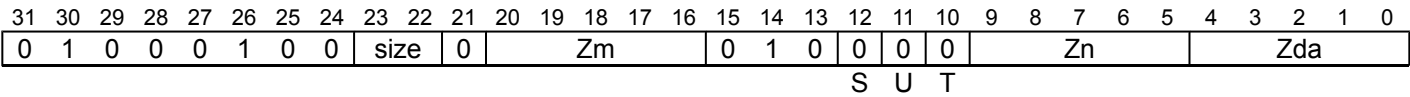
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLALB (vectors)

Signed multiply-add long to accumulator (bottom)

Multiply the corresponding even-numbered signed elements of the first and second source vectors and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.



```
SMLALB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + product;

Z[da, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SMLALT (indexed)

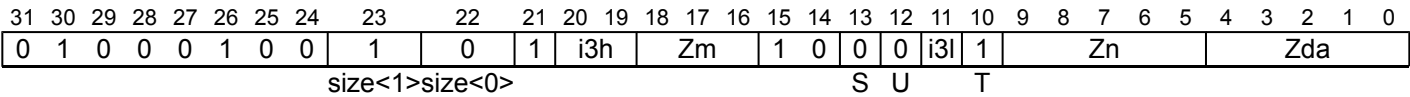
Signed multiply-add long to accumulator (top, indexed)

Multiply the odd-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment and destructively add to the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

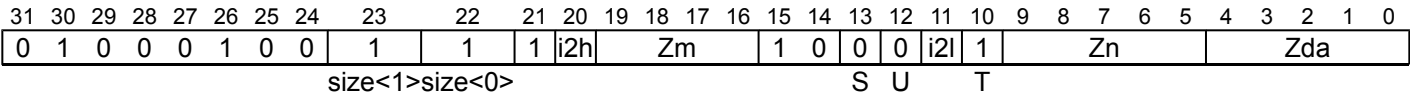
### 32-bit



**SMLALT** <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

### 64-bit



**SMLALT** <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2) < 2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + product;

Z[da, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

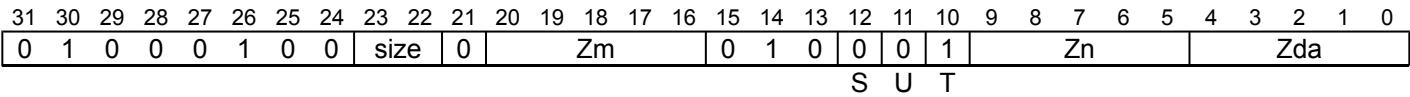
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLALT (vectors)

Signed multiply-add long to accumulator (top)

Multiply the corresponding odd-numbered signed elements of the first and second source vectors and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.



```
SMLALT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + product;

Z[da, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SMLS LB (indexed)

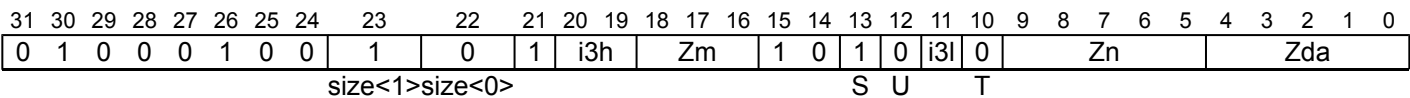
Signed multiply-subtract long from accumulator (bottom, indexed)

Multiply the even-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment and destructively subtract from the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

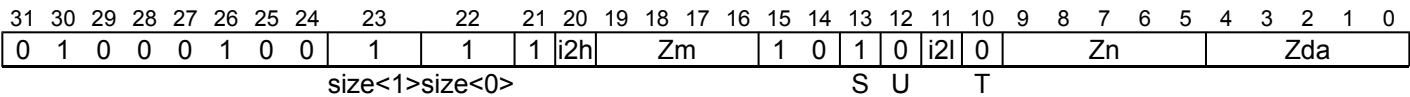
### 32-bit



SMLS LB <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

### 64-bit



SMLS LB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2) < 2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] - product;

Z[da, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is

UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

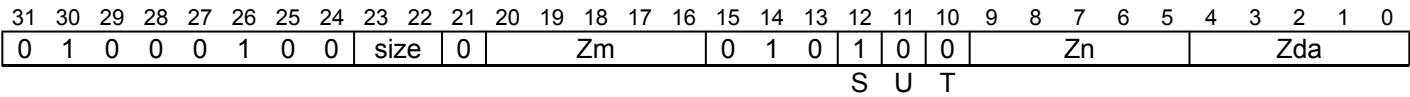
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLS LB (vectors)

Signed multiply-subtract long from accumulator (bottom)

Multiply the corresponding even-numbered signed elements of the first and second source vectors and destructively subtract from the overlapping double-width elements of the addend vector. This instruction is unpredicated.



SMLS LB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] - product;

Z[da, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SMLSLT (indexed)

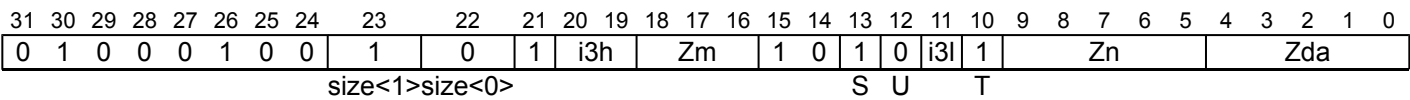
Signed multiply-subtract long from accumulator (top, indexed)

Multiply the odd-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment and destructively subtract from the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

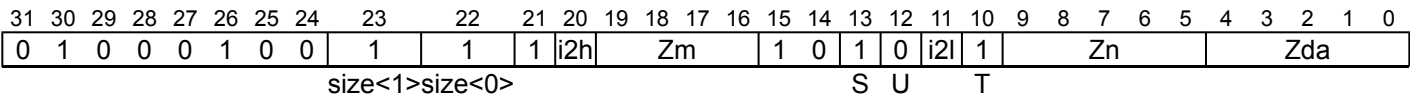
### 32-bit



SMLSLT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

### 64-bit



SMLSLT <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2) < 2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] - product;

Z[da, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

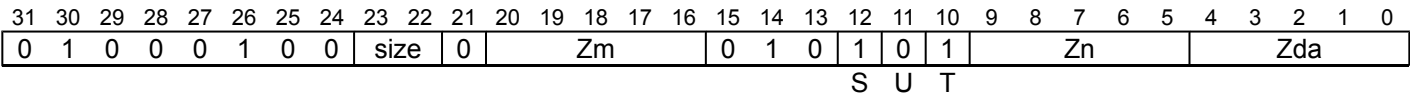
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLSLT (vectors)

Signed multiply-subtract long from accumulator (top)

Multiply the corresponding odd-numbered signed elements of the first and second source vectors and destructively subtract from the overlapping double-width elements of the addend vector. This instruction is unpredicated.



SMLSLT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] - product;

Z[da, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMMLA

Signed integer matrix multiply-accumulate

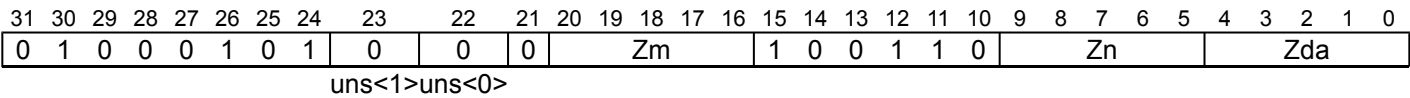
The signed integer matrix multiply-accumulate instruction multiplies the 2×8 matrix of signed 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of signed 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.I8MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE  
(FEAT\_I8MM)



```
SMMLA <Zda>.S, <Zn>.B, <Zm>.B
```

```
if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result = Zeros(VL);
bits(128) op1, op2;
bits(128) res, addend;

for s = 0 to segments-1
    op1 = Elem[operand1, s, 128];
    op2 = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];
    res = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
    Elem[result, s, 128] = res;

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMULH (predicated)

Signed multiply returning high half (predicated)

Widening multiply signed integer values in active elements of the first source vector by corresponding elements of the second source vector and destructively place the high half of the result in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	0	0	1	0	0	0	0	0	Pg	Zm				Zdn							
H																U															

```
SMULH <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if ActivePredicateElement(mask, e, esize) then
        integer product = (element1 * element2) >> esize;
        Elem[result, e, esize] = product<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

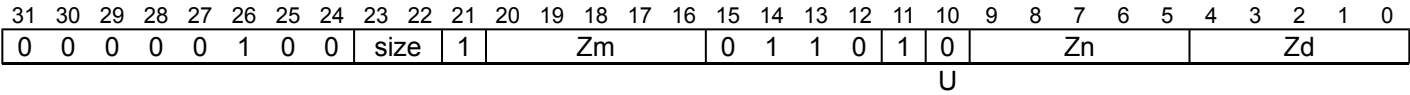
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMULH (unpredicated)

Signed multiply returning high half (unpredicated)

Widening multiply signed integer values of all elements of the first source vector by corresponding elements of the second source vector and place the high half of the result in the corresponding elements of the destination vector. This instruction is unpredicated.



```
SMULH <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    integer product = (element1 * element2) >> esize;
    Elem[result, e, esize] = product<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.





## SMULLB (indexed)

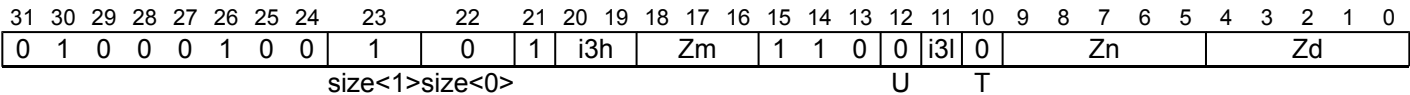
Signed multiply long (bottom, indexed)

Multiply the even-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, and place the results in the overlapping double-width elements of the destination vector register.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

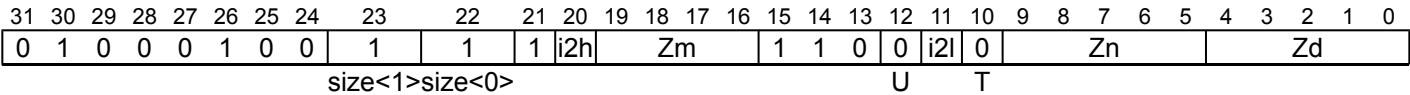
### 32-bit



SMULLB <Zd>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;
```

### 64-bit



SMULLB <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;
```

### Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer res = element1 * element2;
    Elem[result, e, 2*esize] = res<2*esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

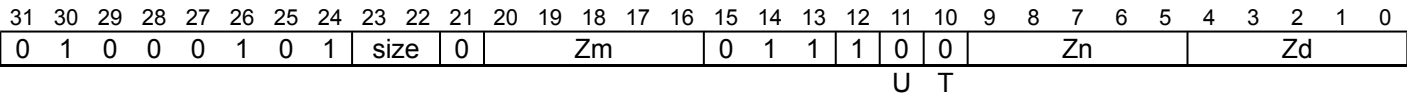
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMULLB (vectors)

Signed multiply long (bottom)

Multiply the corresponding even-numbered signed elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



```
SMULLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    integer res = element1 * element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SMULLT (indexed)

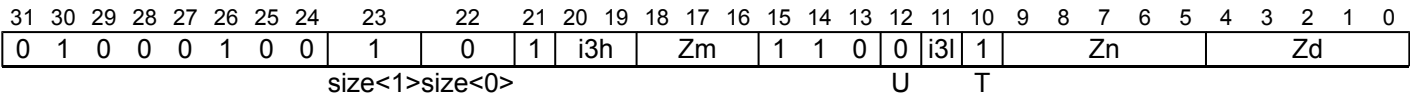
Signed multiply long (top, indexed)

Multiply the odd-numbered signed elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment, and place the results in the overlapping double-width elements of the destination vector register.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

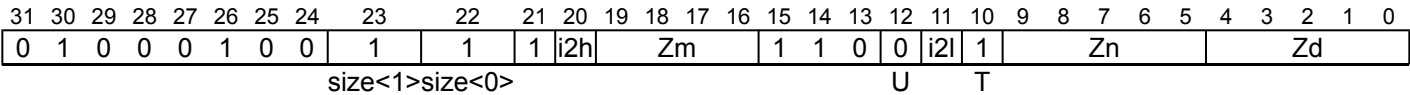
### 32-bit



SMULLT <Zd>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;
```

### 64-bit



SMULLT <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;
```

### Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer res = element1 * element2;
    Elem[result, e, 2*esize] = res<2*esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

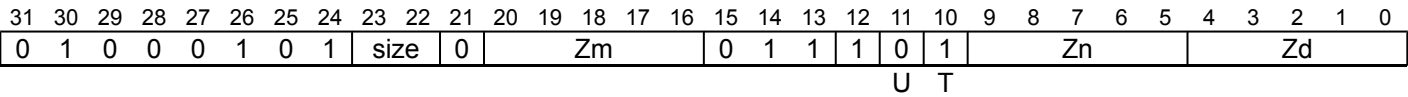
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMULLT (vectors)

Signed multiply long (top)

Multiply the corresponding odd-numbered signed elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



```
SMULLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    integer res = element1 * element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:



- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SPLICE

Splice two vectors under predicate control

Select a region from the first source vector and copy it to the lowest-numbered elements of the result. Then set any remaining elements of the result to a copy of the lowest-numbered elements from the second source vector. The region is selected using the first and last true elements in the vector select predicate register. The result is placed destructively in the destination and first source vector, or constructively in the destination vector.

The Destructive encoding of this instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE: The MOVPRFX instruction must be unpredicated. The MOVPRFX instruction must specify the same destination register as this instruction. The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

It has encodings from 2 classes: [Constructive](#) and [Destructive](#)

Constructive

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	0	1	1	0	1	1	0	0		Pv						Zn				Zd		

SPLICE <Zd>.<T>, <Pv>, { <Zn1>.<T>, <Zn2>.<T> }

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer v = UInt(Pv);
integer dst = UInt(Zd);
integer s1 = UInt(Zn);
integer s2 = (s1 + 1) MOD 32;
```

Destructive

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	0	1	1	0	0	1	0	0		Pv						Zm				Zdn		

SPLICE <Zdn>.<T>, <Pv>, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer v = UInt(Pv);
integer dst = UInt(Zdn);
integer s1 = dst;
integer s2 = UInt(Zm);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

- <Pv> Is the name of the vector select predicate register P0-P7, encoded in the "Pv" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded in the "Zn" field.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[v, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[s1, VL] else Zeros(VL);
bits(VL) operand2 = Z[s2, VL];
bits(VL) result;
integer x = 0;
boolean active = FALSE;
constant integer lastnum = LastActiveElement(mask, esize);

if lastnum >= 0 then
    for e = 0 to lastnum
        active = active || ActivePredicateElement(mask, e, esize);
        if active then
            Elem[result, x, esize] = Elem[operand1, e, esize];
            x = x + 1;

constant integer nelements = (elements - x) - 1;
for e = 0 to nelements
    Elem[result, x, esize] = Elem[operand2, e, esize];
    x = x + 1;

Z[dst, VL] = result;
```

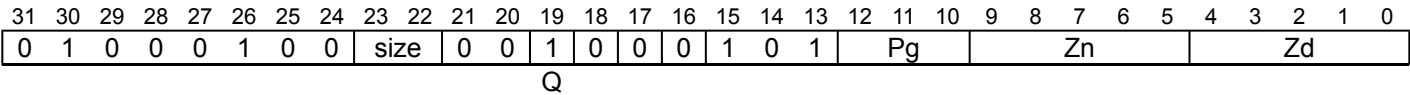
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQABS

Signed saturating absolute value

Compute the absolute value of the signed integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Inactive elements in the destination vector register remain unmodified.



SQABS <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer element = SInt(Elem[operand, e, esize]);
        element = Abs(element);
        Elem[result, e, esize] = SignedSat(element, esize);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

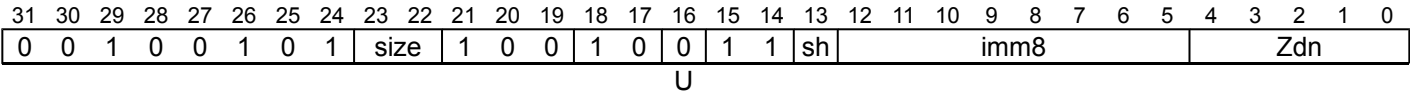


SQADD (immediate)

Signed saturating add immediate (unpredicated)

Signed saturating add of an unsigned immediate to each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.  
The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".



```
SQADD <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + imm, esize, unsigned);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

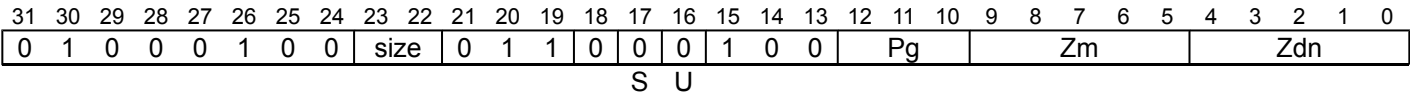
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQADD (vectors, predicated)

Signed saturating addition (predicated)

Add active signed elements of the first source vector to corresponding signed elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Inactive elements in the destination vector register remain unmodified.



SQADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = SInt(Sat(element1 + element2, esize, unsigned));
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```



## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

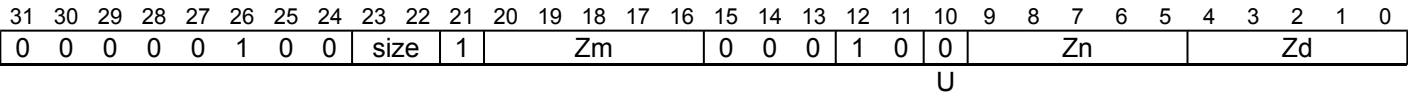
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQADD (vectors, unpredicated)

Signed saturating add vectors (unpredicated)

Signed saturating add all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.



SQADD <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + element2, esize, unsigned);

Z[d, VL] = result;
```

SQCADD

Saturating complex integer add with rotate

Add the real and imaginary components of the integral complex numbers from the first source vector to the complex numbers from the second source vector which have first been rotated by 90 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, equivalent to multiplying the complex numbers in the second source vector by  $\pm j$  beforehand. Destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size	0	0	0	0	0	1	1	1	0	1	1	rot	Zm				Zdn						

SQCADD <Zdn>.<T>, <Zdn>.<T>, <Zm>.<T>, <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
boolean sub_i = (rot == '0');
boolean sub_r = (rot == '1');
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in "rot":

rot	<const>
0	#90
1	#270

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for p = 0 to pairs-1
    integer acc_r = SInt(Elem[operand1, 2 * p + 0, esize]);
    integer acc_i = SInt(Elem[operand1, 2 * p + 1, esize]);
    integer elt2_r = SInt(Elem[operand2, 2 * p + 0, esize]);
    integer elt2_i = SInt(Elem[operand2, 2 * p + 1, esize]);
    if sub_i then
        acc_r = acc_r - elt2_i;
        acc_i = acc_i + elt2_r;
    if sub_r then
        acc_r = acc_r + elt2_i;
        acc_i = acc_i - elt2_r;

    Elem[result, 2 * p + 0, esize] = SignedSat(acc_r, esize);
    Elem[result, 2 * p + 1, esize] = SignedSat(acc_i, esize);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

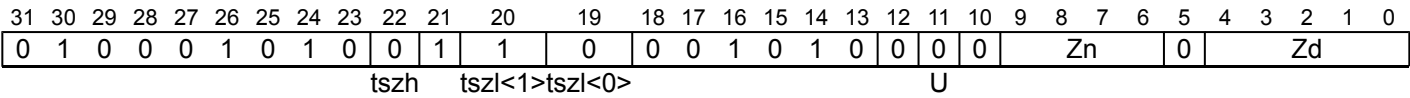
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQCVTN

Signed saturating extract narrow and interleave

Saturate the signed integer value in each element of the group of two source vectors to half the original source element width, and place the two-way interleaved results in the half-width destination elements.  
This instruction is unpredicated.

SVE2  
(FEAT\_SVE2p1)



```
SQCVTN <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;

for e = 0 to elements-1
  for i = 0 to 1
    bits(VL) operand = Z[n+i, VL];
    integer element = SInt(Elem[operand, e, 2 * esize]);
    Elem[result, 2*e + i, esize] = SignedSat(element, esize);
Z[d, VL] = result;
```

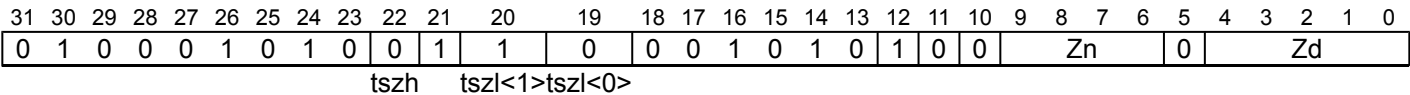
SQCVTUN

Signed saturating unsigned extract narrow and interleave

Saturate the signed integer value in each element of the group of two source vectors to unsigned integer value that is half the original source element width, and place the two-way interleaved results in the half-width destination elements.

This instruction is unpredicated.

SVE2
(FEAT\_SVE2p1)



SQCVTUN <Zd>.H, { <Zn1>.S-<Zn2>.S }

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;

for e = 0 to elements-1
  for i = 0 to 1
    bits(VL) operand = Z[n+i, VL];
    integer element = SInt(Elem[operand, e, 2 * esize]);
    Elem[result, 2*e + i, esize] = UnsignedSat(element, esize);

Z[d, VL] = result;
```

# SQDECB

Signed saturating decrement scalar by multiple of 8-bit predicate constraint element count

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

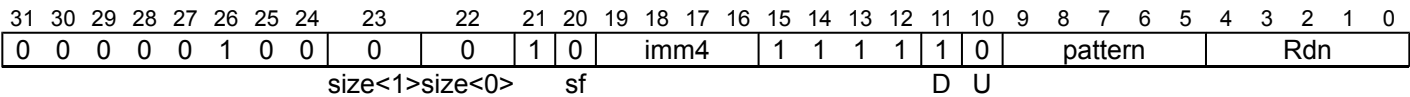
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

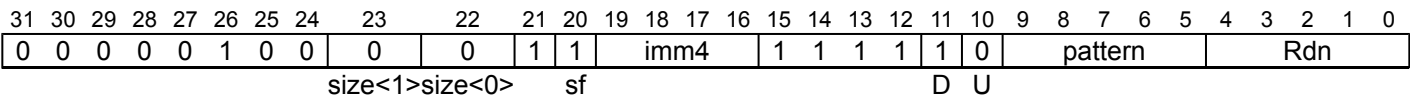
## 32-bit



SQDECB <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
constant integer ssize = 32;
```

## 64-bit



SQDECB <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
constant integer ssize = 64;
```

## Assembler Symbols

- <Xdn>
- Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Wdn>
- Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern>
- Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## SQDECD (scalar)

Signed saturating decrement scalar by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

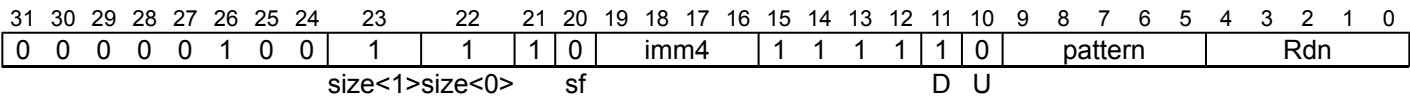
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

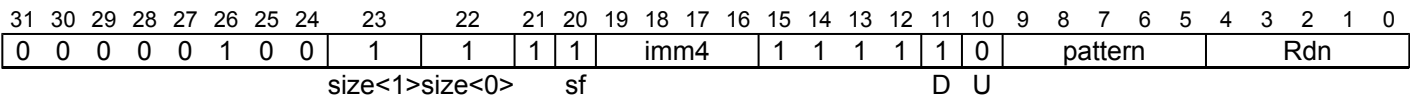
### 32-bit



SQDECD <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
constant integer ssize = 32;
```

### 64-bit



SQDECD <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
constant integer ssize = 64;
```

### Assembler Symbols

- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDECD (vector)

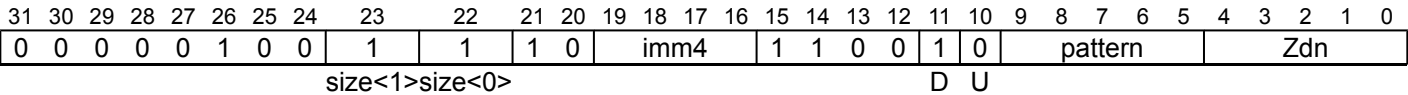
Signed saturating decrement vector by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 64-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



```
SQDECD <Zdn>.D{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# SQDECH (scalar)

Signed saturating decrement scalar by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

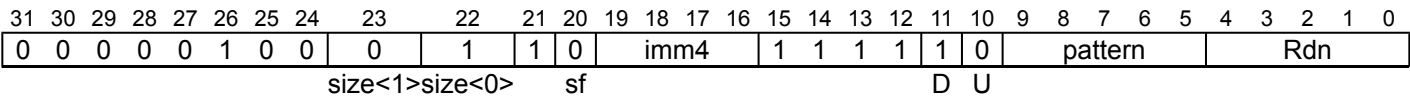
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

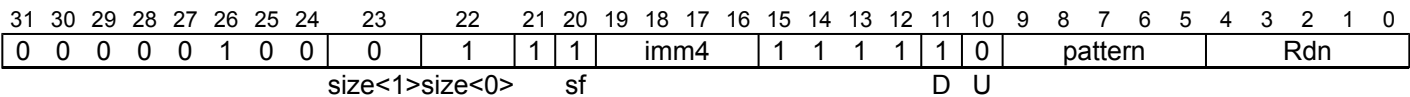
## 32-bit



SQDECH <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
constant integer ssize = 32;
```

## 64-bit



SQDECH <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
constant integer ssize = 64;
```

## Assembler Symbols

- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDECH (vector)

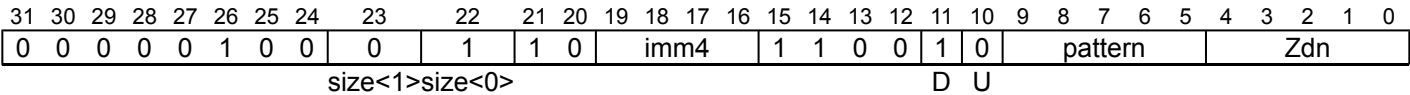
Signed saturating decrement vector by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 16-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



SQDECH <Zdn>.H{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

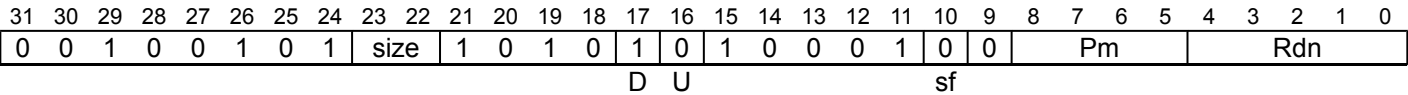


SQDECP (scalar)

Signed saturating decrement scalar by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.  
It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

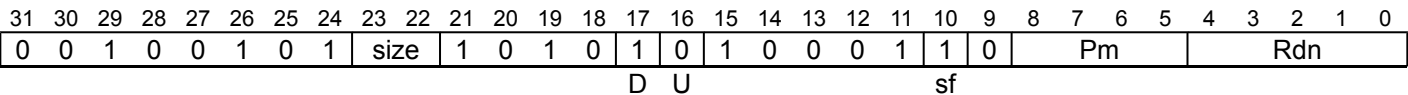
32-bit



SQDECP <Xdn>, <Pm>.<T>, <Wdn>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
constant integer ssize = 32;
```

64-bit



SQDECP <Xdn>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
constant integer ssize = 64;
```

Assembler Symbols

- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D
- <Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(ssize) operand1 = X[dn, ssize];  
bits(PL) operand2 = P[m, PL];  
bits(ssize) result;  
integer count = 0;  
  
for e = 0 to elements-1  
    if ActivePredicateElement(operand2, e, esize) then  
        count = count + 1;  
  
integer element = Int(operand1, unsigned);  
(result, -) = SatQ(element - count, ssize, unsigned);  
X[dn, 64] = Extend(result, 64, unsigned);
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

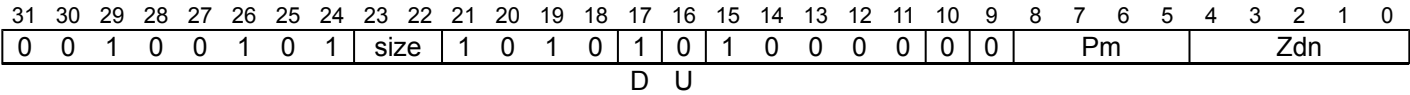
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDECP (vector)

Signed saturating decrement vector by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to decrement all destination vector elements. The results are saturated to the element signed integer range.  
The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.



```
SQDECP <Zdn>.<T>, <Pm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) operand2 = P[m, PL];
bits(VL) result;
integer count = 0;

for e = 0 to elements-1
    if ActivePredicateElement(operand2, e, esize) then
        count = count + 1;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element - count, esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.

- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# SQDECW (scalar)

Signed saturating decrement scalar by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

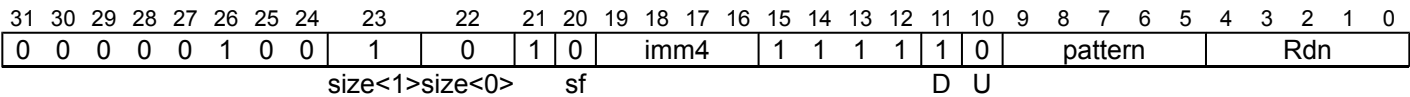
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

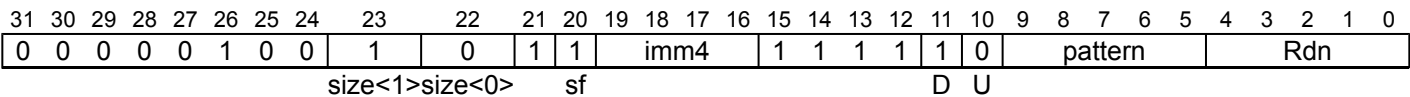
## 32-bit



SQDECW <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
constant integer ssize = 32;
```

## 64-bit



SQDECW <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
constant integer ssize = 64;
```

## Assembler Symbols

- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDECW (vector)

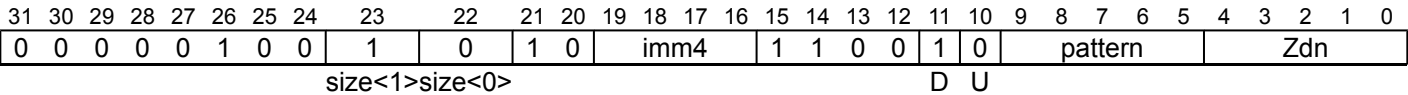
Signed saturating decrement vector by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 32-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



```
SQDECW <Zdn>.S{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## SQDMLALB (indexed)

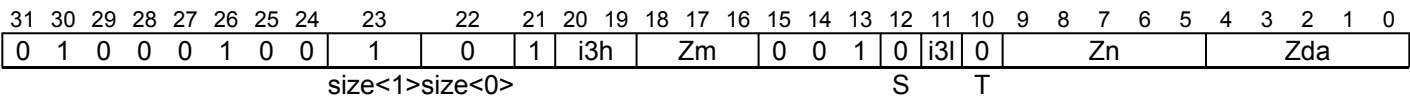
Signed saturating doubling multiply-add long to accumulator (bottom, indexed)

Multiply then double the even-numbered signed elements within each 128-bit segment of the first source vector and specified signed element in the corresponding second source vector segment. Each intermediate value is saturated to the double-width N-bit value's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Then destructively add to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ .

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

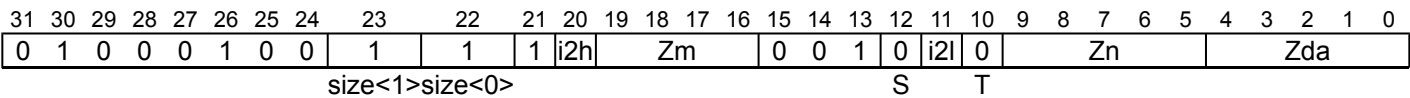
### 32-bit



SQDMLALB <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

### 64-bit



SQDMLALB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer element3 = SInt(Elem[result, e, 2*esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, 2*esize));
    integer res = element3 + product;
    Elem[result, e, 2*esize] = SignedSat(res, 2*esize);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

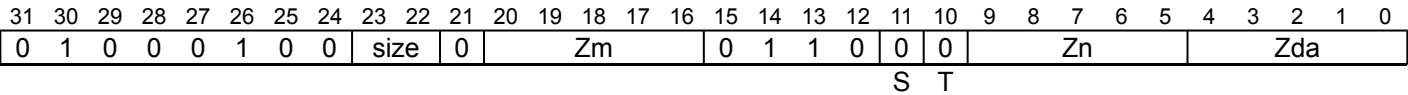
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMLALB (vectors)

Signed saturating doubling multiply-add long to accumulator (bottom)

Multiply then double the corresponding even-numbered signed elements of the first and second source vectors. Each intermediate value is saturated to the double-width N-bit value's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Then destructively add to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.



```
SQDMLALB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel1 = 0;
integer sel2 = 0;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, esize));
    Elem[result, e, esize] = SignedSat(element3 + product, esize);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

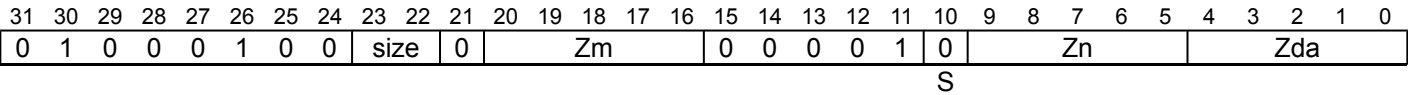
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMLALBT

Signed saturating doubling multiply-add long to accumulator (bottom × top)

Multiply then double the corresponding even-numbered signed elements of the first and odd-numbered signed elements of the second source vector. Each intermediate value is saturated to the double-width N-bit value's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Then destructively add to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.



SQDMLALBT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel1 = 0;
integer sel2 = 1;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, esize));
    Elem[result, e, esize] = SignedSat(element3 + product, esize);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SQDMLALT (indexed)

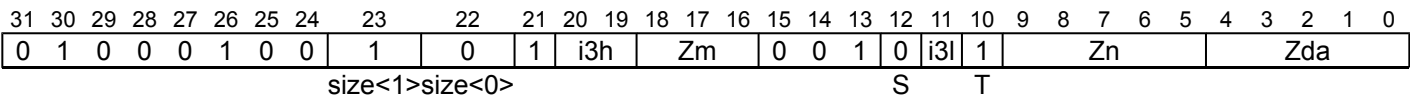
Signed saturating doubling multiply-add long to accumulator (top, indexed)

Multiply then double the odd-numbered signed elements within each 128-bit segment of the first source vector and the specified signed element in the corresponding second source vector segment. Each intermediate value is saturated to the double-width N-bit value's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Then destructively add to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ .

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

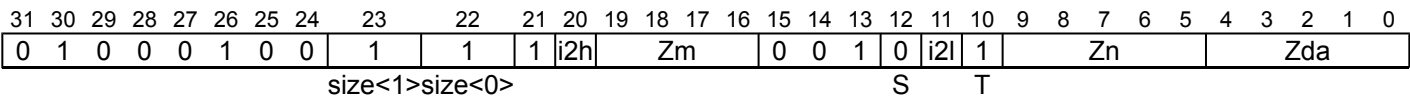
### 32-bit



**SQDMLALT** <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

### 64-bit



**SQDMLALT** <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer element3 = SInt(Elem[result, e, 2*esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, 2*esize));
    integer res = element3 + product;
    Elem[result, e, 2*esize] = SignedSat(res, 2*esize);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

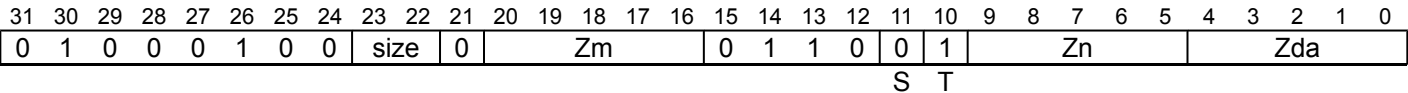
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SQDMLALT (vectors)

Signed saturating doubling multiply-add long to accumulator (top)

Multiply then double the corresponding odd-numbered signed elements of the first and second source vectors. Each intermediate value is saturated to the double-width N-bit value's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Then destructively add to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.



```
SQDMLALT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel1 = 1;
integer sel2 = 1;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, esize));
    Elem[result, e, esize] = SignedSat(element3 + product, esize);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SQDMLSLB (indexed)

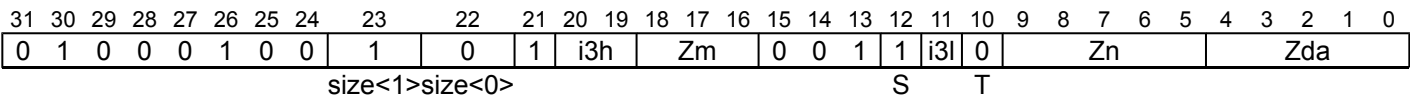
Signed saturating doubling multiply-subtract long from accumulator (bottom, indexed)

Multiply then double the even-numbered signed elements within each 128-bit segment of the first source vector and the specified signed element in the corresponding second source vector segment. Each intermediate value is saturated to the double-width N-bit value's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ .

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

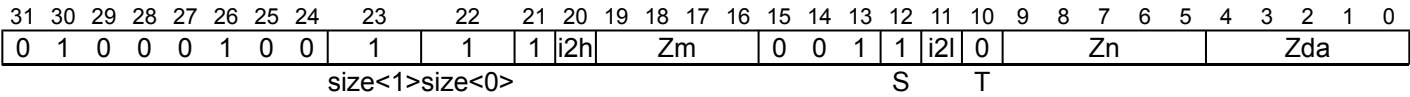
### 32-bit



SQDMLSLB <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

### 64-bit



SQDMLSLB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer element3 = SInt(Elem[result, e, 2*esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, 2*esize));
    integer res = element3 - product;
    Elem[result, e, 2*esize] = SignedSat(res, 2*esize);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

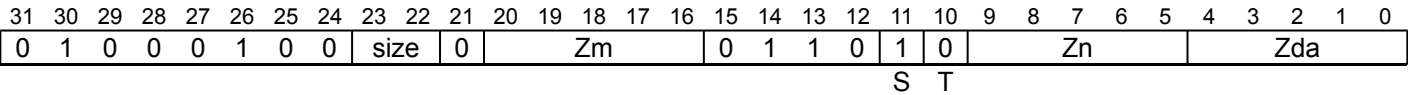
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMLSLB (vectors)

Signed saturating doubling multiply-subtract long from accumulator (bottom)

Multiply then double the corresponding even-numbered signed elements of the first and second source vectors. Each intermediate value is saturated to the double-width N-bit value's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.



SQDMLSLB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel1 = 0;
integer sel2 = 0;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, esize));
    Elem[result, e, esize] = SignedSat(element3 - product, esize);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

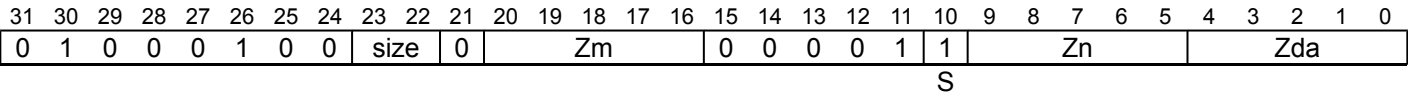
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMLSLBT

Signed saturating doubling multiply-subtract long from accumulator (bottom × top)

Multiply then double the corresponding even-numbered signed elements of the first and odd-numbered signed elements of the second source vector. Each intermediate value is saturated to the double-width N-bit value's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.



```
SQDMLSLBT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel1 = 0;
integer sel2 = 1;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, esize));
    Elem[result, e, esize] = SignedSat(element3 - product, esize);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SQDMLSLT (indexed)

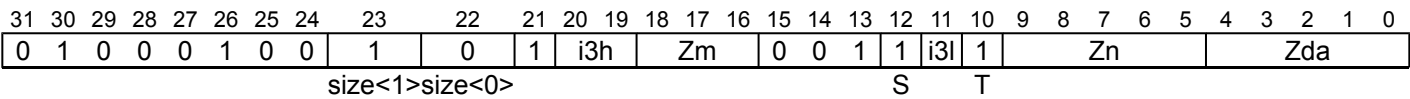
Signed saturating doubling multiply-subtract long from accumulator (top, indexed)

Multiply then double the odd-numbered signed elements within each 128-bit segment of the first source vector and the specified signed element in the corresponding second source vector segment. Each intermediate value is saturated to the double-width N-bit value's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ .

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: 32-bit and 64-bit

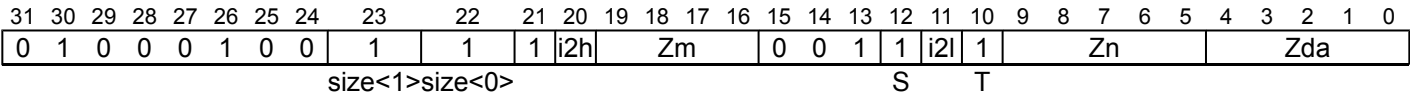
32-bit



SQDMLSLT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

64-bit



SQDMLSLT <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer element3 = SInt(Elem[result, e, 2*esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, 2*esize));
    integer res = element3 - product;
    Elem[result, e, 2*esize] = SignedSat(res, 2*esize);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

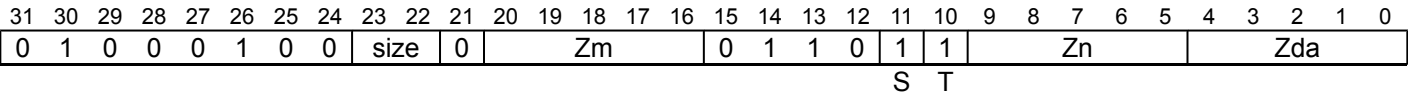
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMLSLT (vectors)

Signed saturating doubling multiply-subtract long from accumulator (top)

Multiply then double the corresponding odd-numbered signed elements of the first and second source vectors. Each intermediate value is saturated to the double-width N-bit value's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.



```
SQDMLSLT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel1 = 1;
integer sel2 = 1;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, esize));
    Elem[result, e, esize] = SignedSat(element3 - product, esize);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SQDMULH (indexed)

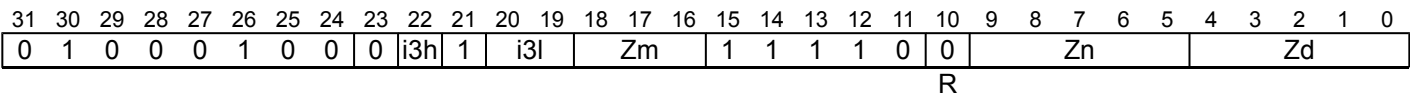
Signed saturating doubling multiply high (indexed)

Multiply all signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, double and place the most significant half of the result in the corresponding elements of the destination vector register. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ .

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element.

It has encodings from 3 classes: [16-bit](#) , [32-bit](#) and [64-bit](#)

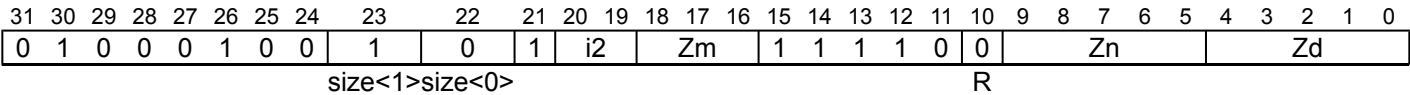
### 16-bit



SQDMULH <Zd>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

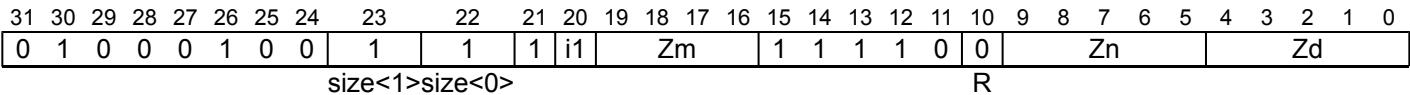
### 32-bit



SQDMULH <Zd>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

### 64-bit



SQDMULH <Zd>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

## Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.  For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, s, esize]);
    integer res = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat(res >> esize, esize);

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

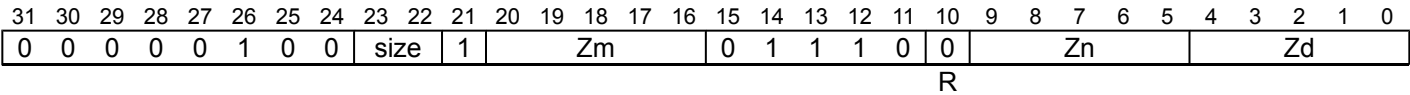
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMULH (vectors)

Signed saturating doubling multiply high (unpredicated)

Multiply then double the corresponding signed elements of the first and second source vectors, and place the most significant half of the results in the corresponding elements of the destination vector register. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.



SQDMULH <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    integer res = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat(res >> esize, esize);

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.





## SQDMULLB (indexed)

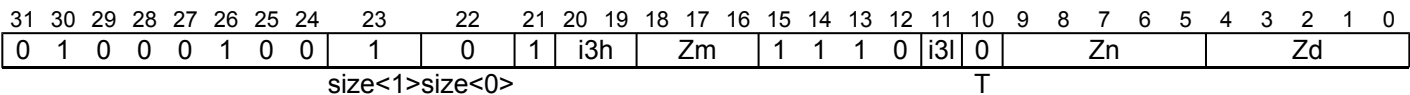
Signed saturating doubling multiply long (bottom, indexed)

Multiply then double the even-numbered signed elements within each 128-bit segment of the first source vector and the specified element in the corresponding second source vector segment, and place the results in overlapping double-width elements of the destination vector register. Each result element is saturated to the double-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ .

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

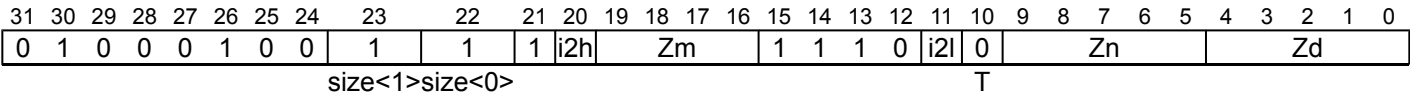
### 32-bit



SQDMULLB <Zd>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;
```

### 64-bit



SQDMULLB <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;
```

### Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer res = 2 * element1 * element2;
    Elem[result, e, 2*esize] = SignedSat(res, 2*esize);

Z[d, VL] = result;
```

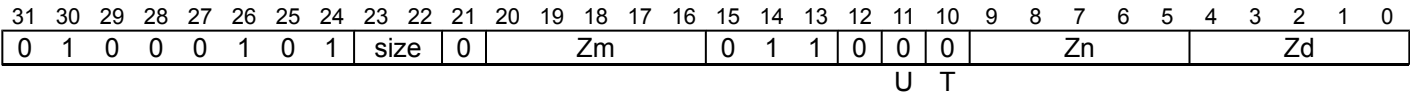
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMULLB (vectors)

Signed saturating doubling multiply long (bottom)

Multiply the corresponding even-numbered signed elements of the first and second source vectors, double and place the results in the overlapping double-width elements of the destination vector. Each result element is saturated to the double-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.



SQDMULLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    integer res = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat(res, esize);

Z[d, VL] = result;
```



## SQDMULLT (indexed)

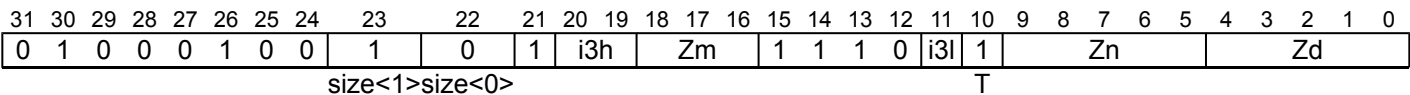
Signed saturating doubling multiply long (top, indexed)

Multiply then double the odd-numbered signed elements within each 128-bit segment of the first source vector and the specified element in the corresponding second source vector segment, and place the results in overlapping double-width elements of the destination vector register. Each result element is saturated to the double-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ .

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

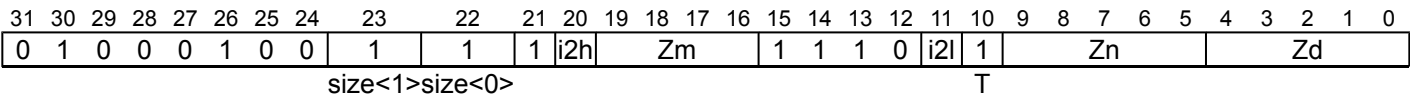
### 32-bit



SQDMULLT <Zd>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;
```

### 64-bit



SQDMULLT <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;
```

### Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer res = 2 * element1 * element2;
    Elem[result, e, 2*esize] = SignedSat(res, 2*esize);

Z[d, VL] = result;
```

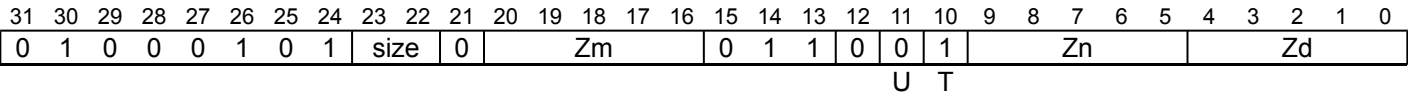
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMULLT (vectors)

Signed saturating doubling multiply long (top)

Multiply the corresponding odd-numbered signed elements of the first and second source vectors, double and place the results in the overlapping double-width elements of the destination vector. Each result element is saturated to the double-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.



```
SQDMULLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    integer res = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat(res, esize);

Z[d, VL] = result;
```





# SQINCB

Signed saturating increment scalar by multiple of 8-bit predicate constraint element count

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

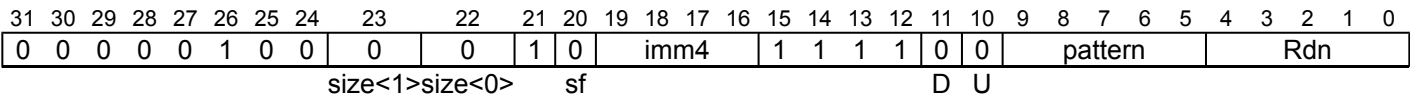
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

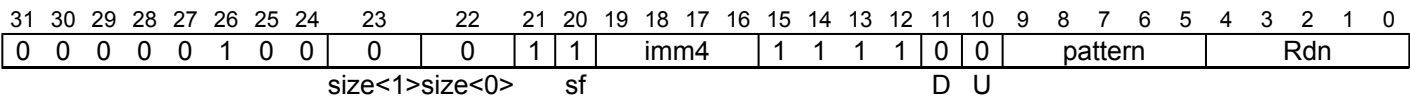
## 32-bit



SQINCB <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
constant integer ssize = 32;
```

## 64-bit



SQINCB <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
constant integer ssize = 64;
```

## Assembler Symbols

- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# SQINCD (scalar)

Signed saturating increment scalar by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

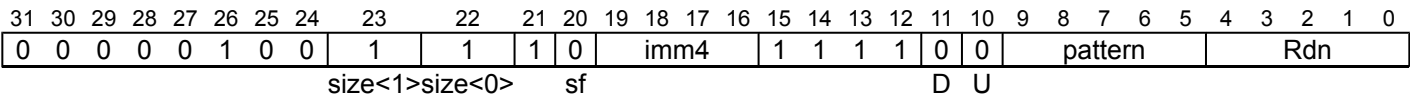
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

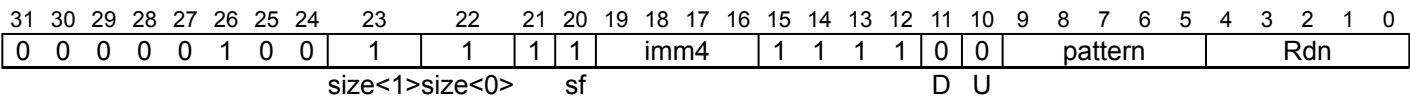
## 32-bit



SQINCD <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
constant integer ssize = 32;
```

## 64-bit



SQINCD <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
constant integer ssize = 64;
```

## Assembler Symbols

- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQINCD (vector)

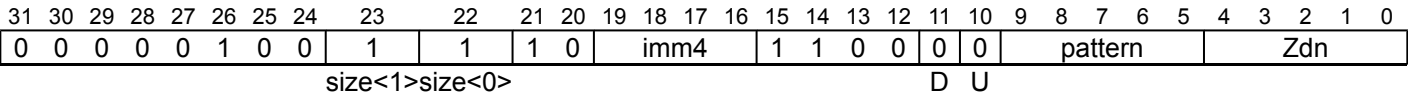
Signed saturating increment vector by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 64-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



```
SQINCD <Zdn>.D{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SQINCH (scalar)

Signed saturating increment scalar by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

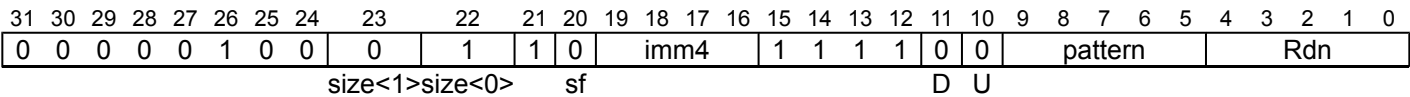
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

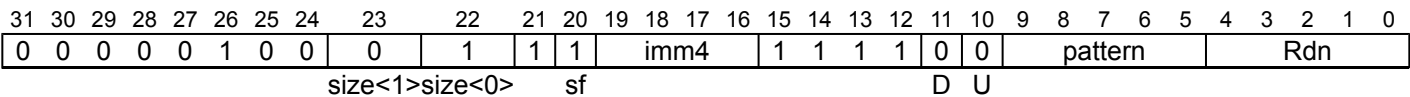
### 32-bit



SQINCH <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
constant integer ssize = 32;
```

### 64-bit



SQINCH <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
constant integer ssize = 64;
```

### Assembler Symbols

- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SQINCH (vector)

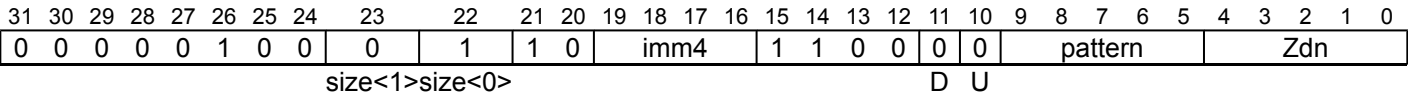
Signed saturating increment vector by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 16-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



```
SQINCH <Zdn>.H{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

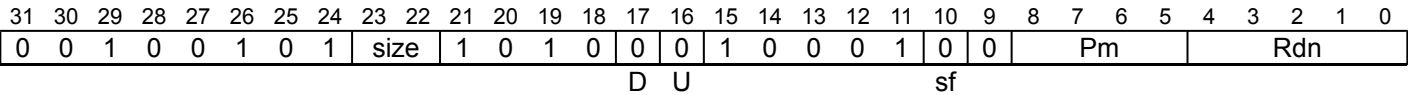
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQINCP (scalar)

Signed saturating increment scalar by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.  
It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

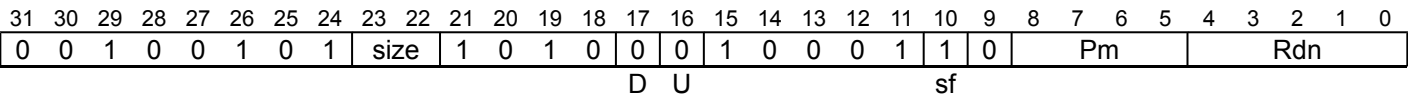
32-bit



SQINCP <Xdn>, <Pm>.<T>, <Wdn>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
constant integer ssize = 32;
```

64-bit



SQINCP <Xdn>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
constant integer ssize = 64;
```

Assembler Symbols

- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(ssize) operand1 = X[dn, ssize];  
bits(PL) operand2 = P[m, PL];  
bits(ssize) result;  
integer count = 0;  
  
for e = 0 to elements-1  
    if ActivePredicateElement(operand2, e, esize) then  
        count = count + 1;  
  
integer element = Int(operand1, unsigned);  
(result, -) = SatQ(element + count, ssize, unsigned);  
X[dn, 64] = Extend(result, 64, unsigned);
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

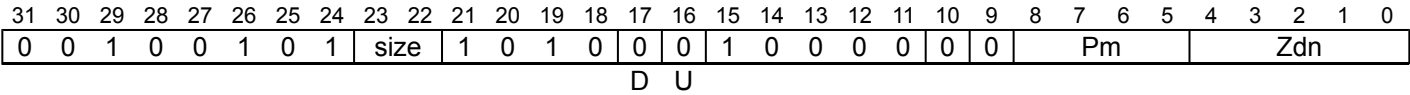
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQINCP (vector)

Signed saturating increment vector by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to increment all destination vector elements. The results are saturated to the element signed integer range. The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.



SQINCP <Zdn>.<T>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) operand2 = P[m, PL];
bits(VL) result;
integer count = 0;

for e = 0 to elements-1
    if ActivePredicateElement(operand2, e, esize) then
        count = count + 1;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element + count, esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.

- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# SQINCW (scalar)

Signed saturating increment scalar by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

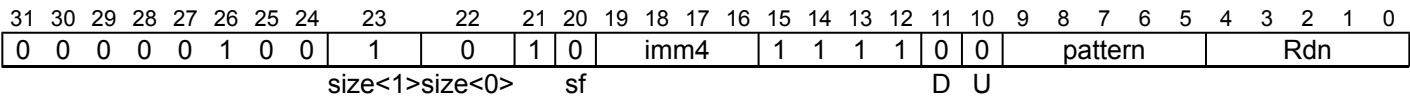
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

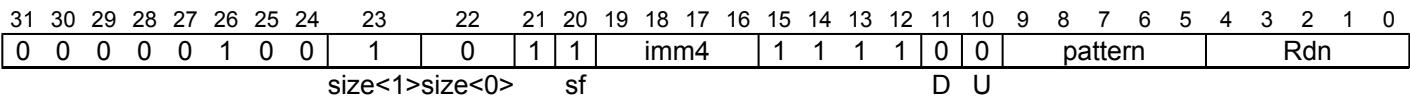
## 32-bit



SQINCW <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
constant integer ssize = 32;
```

## 64-bit



SQINCW <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
constant integer ssize = 64;
```

## Assembler Symbols

- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SQINCW (vector)

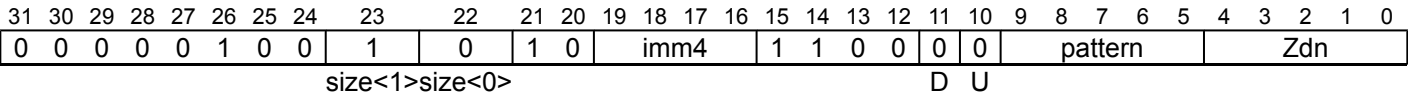
Signed saturating increment vector by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 32-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



```
SQINCW <Zdn>.S{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

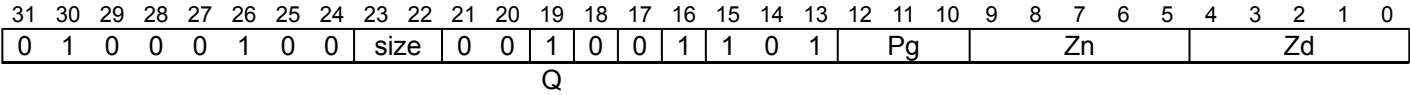
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQNEG

Signed saturating negate

Negate the signed integer value in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Inactive elements in the destination vector register remain unmodified.



SQNEG <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    integer element = SInt(Elem[operand, e, esize]);
    element = -element;
    Elem[result, e, esize] = SignedSat(element, esize);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# SQRDCMLAH (indexed)

Saturating rounding doubling complex integer multiply-add high with rotate (indexed)

Multiply without saturation the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in each 128-bit segment of the first source vector by the specified complex number in the corresponding the second source vector segment rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation. Then double and add the products to the corresponding components of the complex numbers in the addend vector. Destructively place the most significant rounded half of the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

It has encodings from 2 classes: [16-bit](#) and [32-bit](#)

## 16-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	0	1	i2		Zm			0	1	1	1	rot				Zn					Zda		
size<1>size<0>																															

SQRDCMLAH <Zda>.H, <Zn>.H, <Zm>.H[<imm>], <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');
```

## 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	1	1	i1	Zm			0			1	1	1	rot			Zn			Zda				
size<1>size<0>																															

SQRDCMLAH <Zda>.S, <Zn>.S, <Zm>.S[<imm>], <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');
```

## Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

- <Zm> For the 16-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 32-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 16-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.  
For the 32-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.
- <const> Is the const specifier, encoded in "rot":

rot	<const>
00	#0
01	#90
10	#180
11	#270

## Operation

```

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
constant integer pairspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

integer res_r, res_i;

for p = 0 to pairs-1
    integer segmentbase = p - (p MOD pairspersegment);
    integer s = segmentbase + index;
    integer elt1_a = SInt(Elem[operand1, 2 * p + sel_a, esize]);
    integer elt2_a = SInt(Elem[operand2, 2 * s + sel_a, esize]);
    integer elt2_b = SInt(Elem[operand2, 2 * s + sel_b, esize]);
    bits(esize) elt3_r = Elem[operand3, 2 * p + 0, esize];
    bits(esize) elt3_i = Elem[operand3, 2 * p + 1, esize];
    integer product_r = elt1_a * elt2_a;
    integer product_i = elt1_a * elt2_b;
    if sub_r then
        res_r = (SInt(elt3_r) << esize) - 2 * product_r;
    else
        res_r = (SInt(elt3_r) << esize) + 2 * product_r;
    res_r = (res_r + (1 << (esize-1))) >> esize;
    Elem[result, 2 * p + 0, esize] = SignedSat(res_r, esize);
    if sub_i then
        res_i = (SInt(elt3_i) << esize) - 2 * product_i;
    else
        res_i = (SInt(elt3_i) << esize) + 2 * product_i;
    res_i = (res_i + (1 << (esize-1))) >> esize;
    Elem[result, 2 * p + 1, esize] = SignedSat(res_i, esize);

Z[da, VL] = result;

```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



SQRDCMLAH (vectors)

Saturating rounding doubling complex integer multiply-add high with rotate

Multiply without saturation the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in the first source vector by the corresponding complex number in the second source vector rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation. Then double and add the products to the corresponding components of the complex numbers in the addend vector. Destructively place the most significant rounded half of the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		0	Zm				0	0	1	1	rot		Zn				Zda						

SQRDCMLAH <Zda>.<T>, <Zn>.<T>, <Zm>.<T>, <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in "rot":

rot	<const>
00	#0
01	#90
10	#180
11	#270



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

integer res_r, res_i;

for p = 0 to pairs-1
    integer elt1_a = SInt(Elem[operand1, 2 * p + sel_a, esize]);
    integer elt2_a = SInt(Elem[operand2, 2 * p + sel_a, esize]);
    integer elt2_b = SInt(Elem[operand2, 2 * p + sel_b, esize]);
    bits(esize) elt3_r = Elem[operand3, 2 * p + 0, esize];
    bits(esize) elt3_i = Elem[operand3, 2 * p + 1, esize];
    integer product_r = elt1_a * elt2_a;
    integer product_i = elt1_a * elt2_b;
    if sub_r then
        res_r = (SInt(elt3_r) << esize) - 2 * product_r;
    else
        res_r = (SInt(elt3_r) << esize) + 2 * product_r;
    res_r = (res_r + (1 << (esize-1))) >> esize;
    Elem[result, 2 * p + 0, esize] = SignedSat(res_r, esize);
    if sub_i then
        res_i = (SInt(elt3_i) << esize) - 2 * product_i;
    else
        res_i = (SInt(elt3_i) << esize) + 2 * product_i;
    res_i = (res_i + (1 << (esize-1))) >> esize;
    Elem[result, 2 * p + 1, esize] = SignedSat(res_i, esize);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SQRDMLAH (indexed)

Signed saturating rounding doubling multiply-add high to accumulator (indexed)

Multiply then double all signed elements within each 128-bit segment of the first source vector and the specified signed element of the corresponding second source vector segment, and destructively add the rounded high half of each result to the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ .

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element.

It has encodings from 3 classes: [16-bit](#) , [32-bit](#) and [64-bit](#)

### 16-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	0	i3h	1	i3l		Zm		0	0	0	1	0	0											
																					S		Zn					Zda			

**SQRDMLAH** <Zda>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	0	1	i2		Zm		0	0	0	1	0	0											
size<1>size<0>																					S		Zn					Zda			

**SQRDMLAH** <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### 64-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	1	1	i1		Zm		0	0	0	1	0	0											
size<1>size<0>																					S		Zn					Zda			

**SQRDMLAH** <Zda>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

## Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.  For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, s, esize]);
    integer element3 = SInt(Elem[operand3, e, esize]);
    integer res = (element3 << esize) + (2 * element1 * element2);
    Elem[result, e, esize] = SignedSat((res + (1 << (esize - 1))) >> esize, esize);

Z[da, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

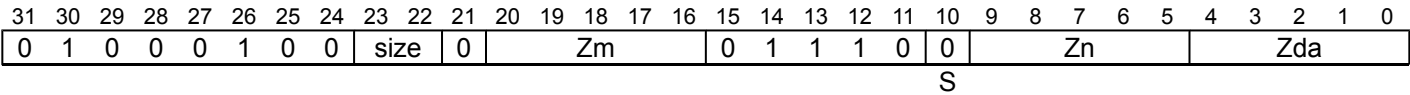
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQRDMLAH (vectors)

Signed saturating rounding doubling multiply-add high to accumulator (unpredicated)

Multiply then double the corresponding signed elements of the first and second source vectors, and destructively add the rounded high half of each result to the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.



```
SQRDMLAH <Zda>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    integer element3 = SInt(Elem[operand3, e, esize]);
    integer res = (element3 << esize) + (2 * element1 * element2);
    Elem[result, e, esize] = SignedSat((res + (1 << (esize - 1))) >> esize, esize);

Z[da, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SQRDMLSH (indexed)

Signed saturating rounding doubling multiply-subtract high from accumulator (indexed)

Multiply then double all signed elements within each 128-bit segment of the first source vector and the specified signed element of the corresponding second source vector segment, and destructively subtract the rounded high half of each result to the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ .

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element.

It has encodings from 3 classes: [16-bit](#) , [32-bit](#) and [64-bit](#)

### 16-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	0	i3h	1	i3l		Zm		0	0	0	1	0	1				Zn					Zda		
																					S										

SQRDMLSH <Zda>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	0	1	i2		Zm		0	0	0	1	0	1				Zn					Zda		
size<1>size<0>											S																				

SQRDMLSH <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### 64-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	1	1	i1	Zm			0			0	0	1	0	1	Zn			Zda					
size<1>size<0>												S																			

SQRDMLSH <Zda>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

## Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.  For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, s, esize]);
    integer element3 = SInt(Elem[operand3, e, esize]);
    integer res = (element3 << esize) - (2 * element1 * element2);
    Elem[result, e, esize] = SignedSat((res + (1 << (esize - 1))) >> esize, esize);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

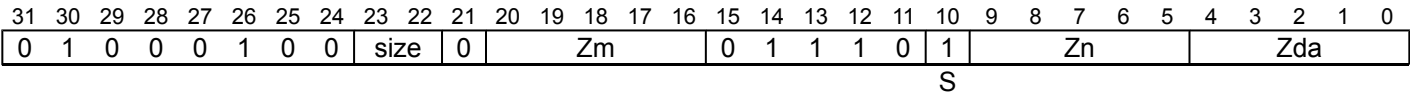
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQRDMLSH (vectors)

Signed saturating rounding doubling multiply-subtract high from accumulator (unpredicated)

Multiply then double the corresponding signed elements of the first and second source vectors, and destructively subtract the rounded high half of each result from the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.



SQRDMLSH <Zda>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    integer element3 = SInt(Elem[operand3, e, esize]);
    integer res = (element3 << esize) - (2 * element1 * element2);
    Elem[result, e, esize] = SignedSat((res + (1 << (esize - 1))) >> esize, esize);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.



- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SQRDMULH (indexed)

Signed saturating rounding doubling multiply high (indexed)

Multiply all signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, double and place the most significant rounded half of the result in the corresponding elements of the destination vector register. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ .

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element.

It has encodings from 3 classes: [16-bit](#) , [32-bit](#) and [64-bit](#)

### 16-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	0	i3h	1	i3l		Zm			1	1	1	1	0	1										
																					R										

**SQRDMULH** <Zd>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

### 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	0	1	i2		Zm			1	1	1	1	0	1										
size<1>size<0>											R																				

**SQRDMULH** <Zd>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

### 64-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	1	1	i1	Zm			1	1	1	1	0	1	Zn			Zd							
size<1>size<0>											R																				

**SQRDMULH** <Zd>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

## Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.  For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, s, esize]);
    integer res = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat((res + (1 << (esize - 1))) >> esize, esize);

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

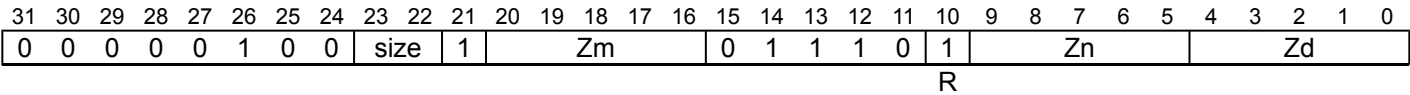
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQRDMULH (vectors)

Signed saturating rounding doubling multiply high (unpredicated)

Multiply then double the corresponding signed elements of the first and second source vectors, and place the most significant rounded half of the result in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.



SQRDMULH <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    integer res = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat((res + (1 << (esize - 1))) >> esize, esize);

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

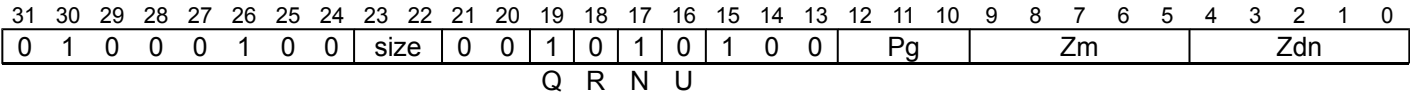
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



SQRSHL

Signed saturating rounding shift left by vector (predicated)

Shift active signed elements of the first source vector by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Inactive elements in the destination vector register remain unmodified.



```
SQRSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    integer element = SInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer res;
    if shift >= 0 then
      res = element << shift;
    else
      shift = -shift;
      res = (element + (1 << (shift - 1))) >> shift;
    Elem[result, e, esize] = SignedSat(res, esize);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

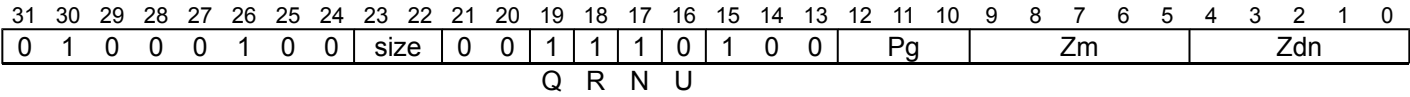
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQRSHLR

Signed saturating rounding shift left reversed vectors (predicated)

Shift active signed elements of the second source vector by corresponding elements of the first source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Inactive elements in the destination vector register remain unmodified.



SQRSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand2 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    integer element = SInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer res;
    if shift >= 0 then
      res = element << shift;
    else
      shift = -shift;
      res = (element + (1 << (shift - 1))) >> shift;
    Elem[result, e, esize] = SignedSat(res, esize);
  else
    Elem[result, e, esize] = Elem[operand2, e, esize];

Z[dn, VL] = result;
```



## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

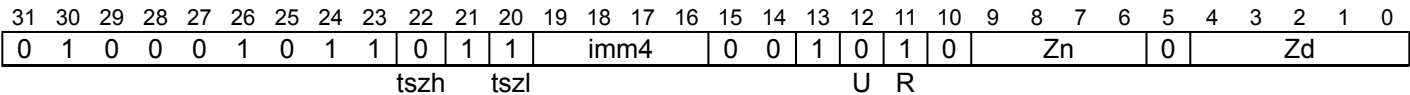
SQRSHRN

Signed saturating rounding shift right narrow by immediate and interleave

Shift right by an immediate value, the signed integer value in each element of the group of two source vectors and place the two-way interleaved rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SVE2
(FEAT\_SVE2p1)



SQRSHRN <Zd>.H, { <Zn1>.S-<Zn2>.S }, #<const>

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
integer shift = esize - UInt(imm4);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <const> Is the immediate shift amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;

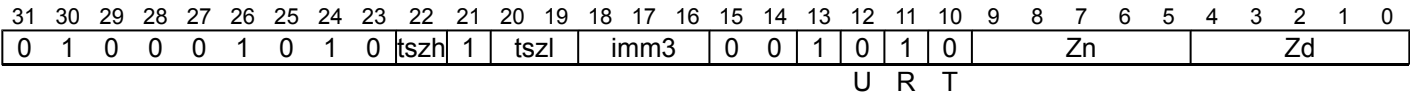
for e = 0 to elements-1
  for i = 0 to 1
    bits(VL) operand = Z[n+i, VL];
    bits(2 * esize) element = Elem[operand, e, 2 * esize];
    integer res = (SInt(element) + (1 << (shift-1))) >> shift;
    Elem[result, 2*e + i, esize] = SignedSat(res, esize);

Z[d, VL] = result;
```

SQRSHRNB

Signed saturating rounding shift right narrow by immediate (bottom)

Shift each signed integer value in the source vector elements right by an immediate value, and place the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



```
SQRSHRNB <Zd>.<T>, <Zn>.<Tb>, #<const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = (SInt(element) + (1 << (shift-1))) >> shift;
    Elem[result, 2*e + 0, esize] = SignedSat(res, esize);
    Elem[result, 2*e + 1, esize] = Zeros(esize);

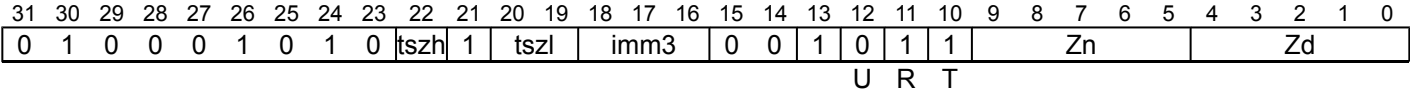
Z[d, VL] = result;
```



SQRSHRNT

Signed saturating rounding shift right narrow by immediate (top)

Shift each signed integer value in the source vector elements right by an immediate value, and place the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



SQRSHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = (SInt(element) + (1 << (shift-1))) >> shift;
    Elem[result, 2*e + 1, esize] = SignedSat(res, esize);

Z[d, VL] = result;
```



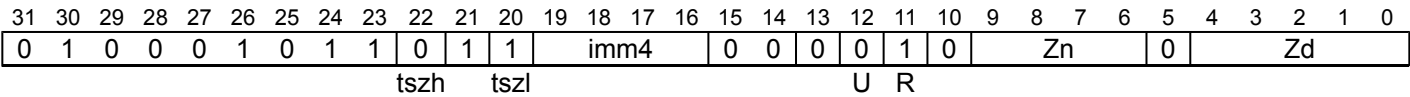
SQRSHRUN

Signed saturating rounding shift right unsigned narrow by immediate and interleave

Shift right by an immediate value, the signed integer value in each element of the group of two source vectors and place the two-way interleaved rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SVE2
(FEAT\_SVE2p1)



SQRSHRUN <Zd>.H, { <Zn1>.S-<Zn2>.S }, #<const>

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
integer shift = esize - UInt(imm4);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <const> Is the immediate shift amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;

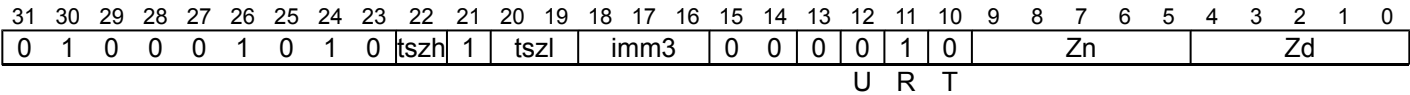
for e = 0 to elements-1
  for i = 0 to 1
    bits(VL) operand = Z[n+i, VL];
    bits(2 * esize) element = Elem[operand, e, 2 * esize];
    integer res = (SInt(element) + (1 << (shift-1))) >> shift;
    Elem[result, 2*e + i, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```

SQRSHRUNB

Signed saturating rounding shift right unsigned narrow by immediate (bottom)

Shift each signed integer value in the source vector elements right by an immediate value, and place the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



SQRSHRUNB <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = (SInt(element) + (1 << (shift-1))) >> shift;
    Elem[result, 2*e + 0, esize] = UnsignedSat(res, esize);
    Elem[result, 2*e + 1, esize] = Zeros(esize);

Z[d, VL] = result;
```

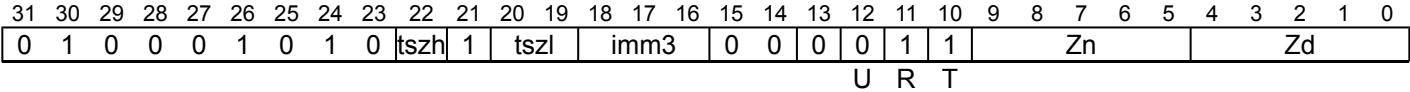




SQRSHRUNT

Signed saturating rounding shift right unsigned narrow by immediate (top)

Shift each signed integer value in the source vector elements right by an immediate value, and place the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



SQRSHRUNT <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = (SInt(element) + (1 << (shift-1))) >> shift;
    Elem[result, 2*e + 1, esize] = UnsignedSat(res, esize);

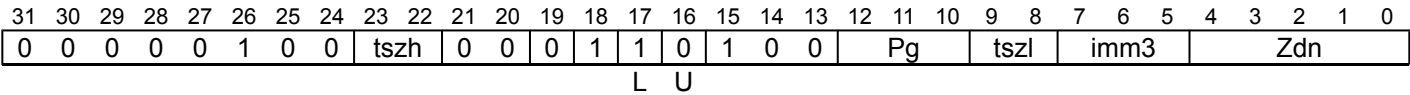
Z[d, VL] = result;
```



SQSHL (immediate)

Signed saturating shift left by immediate

Shift left by immediate each active signed element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.



SQSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = element1 << shift;
        Elem[result, e, esize] = SignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

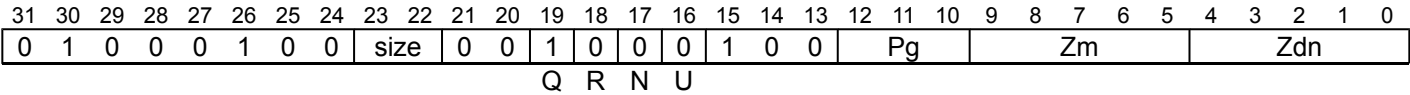
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQSHL (vectors)

Signed saturating shift left by vector (predicated)

Shift active signed elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Inactive elements in the destination vector register remain unmodified.



SQSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    integer element = SInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer res;
    if shift >= 0 then
      res = element << shift;
    else
      shift = -shift;
      res = element >> shift;
    Elem[result, e, esize] = SignedSat(res, esize);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQSHLR

Signed saturating shift left reversed vectors (predicated)

Shift active signed elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	0	0	1	1	0	0	1	0	0	Pg				Zm				Zdn					
Q R N U																															

SQSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand2 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    integer element = SInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer res;
    if shift >= 0 then
      res = element << shift;
    else
      shift = -shift;
      res = element >> shift;
    Elem[result, e, esize] = SignedSat(res, esize);
  else
    Elem[result, e, esize] = Elem[operand2, e, esize];

Z[dn, VL] = result;
```



## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

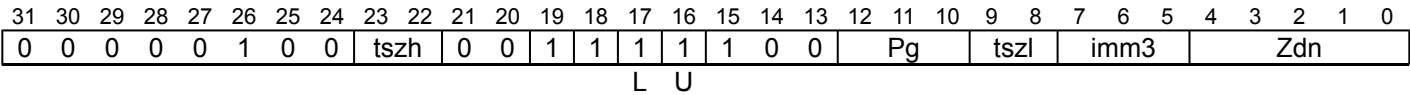
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQSHLU

Signed saturating shift left unsigned by immediate

Shift left by immediate each active signed element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.



SQSHLU <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = element1 << shift;
        Elem[result, e, esize] = UnsignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

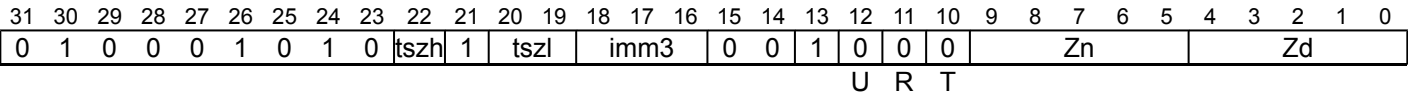
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQSHRNB

Signed saturating shift right narrow by immediate (bottom)

Shift each signed integer value in the source vector elements right by an immediate value, and place the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



```
SQSHRNB <Zd>.<T>, <Zn>.<Tb>, #<const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = SInt(element) >> shift;
    Elem[result, 2*e + 0, esize] = SignedSat(res, esize);
    Elem[result, 2*e + 1, esize] = Zeros(esize);

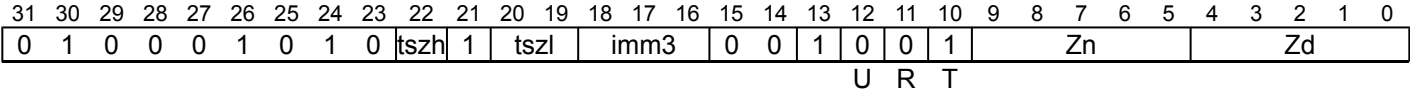
Z[d, VL] = result;
```



SQSHRNT

Signed saturating shift right narrow by immediate (top)

Shift each signed integer value in the source vector elements right by an immediate value, and place the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



SQSHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = SInt(element) >> shift;
    Elem[result, 2*e + 1, esize] = SignedSat(res, esize);

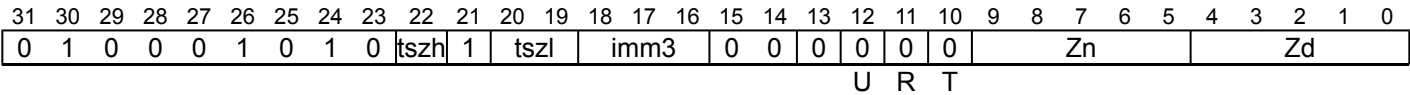
Z[d, VL] = result;
```



SQSHRUNB

Signed saturating shift right unsigned narrow by immediate (bottom)

Shift each signed integer value in the source vector elements right by an immediate value, and place the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



```
SQSHRUNB <Zd>.<T>, <Zn>.<Tb>, #<const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = SInt(element) >> shift;
    Elem[result, 2*e + 0, esize] = UnsignedSat(res, esize);
    Elem[result, 2*e + 1, esize] = Zeros(esize);

Z[d, VL] = result;
```

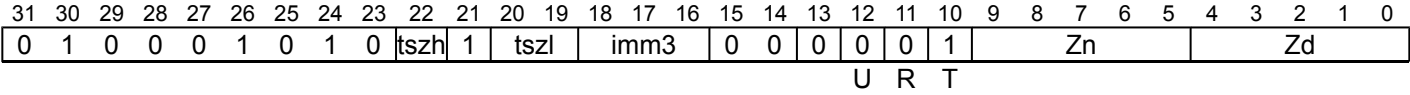




SQSHRUNT

Signed saturating shift right unsigned narrow by immediate (top)

Shift each signed integer value in the source vector elements right by an immediate value, and place the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



```
SQSHRUNT <Zd>.<T>, <Zn>.<Tb>, #<const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = SInt(element) >> shift;
    Elem[result, 2*e + 1, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```



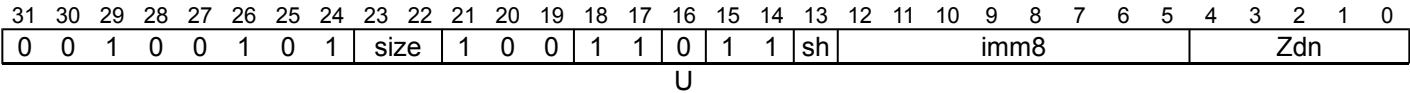
SQSUB (immediate)

Signed saturating subtract immediate (unpredicated)

Signed saturating subtract of an unsigned immediate from each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".



SQSUB <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - imm, esize, unsigned);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

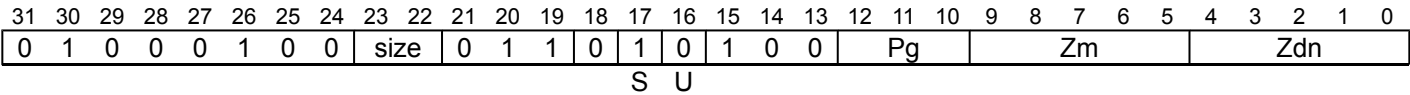
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQSUB (vectors, predicated)

Signed saturating subtraction (predicated)

Subtract active signed elements of the second source vector from corresponding signed elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Inactive elements in the destination vector register remain unmodified.



SQSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = SInt(Sat(element1 - element2, esize, unsigned));
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

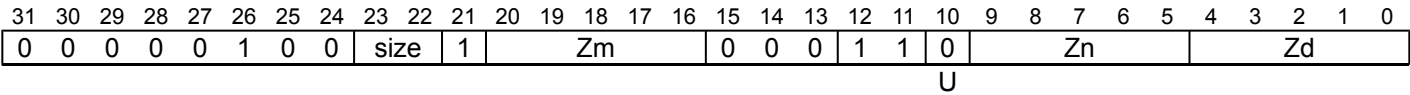
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQSUB (vectors, unpredicated)

Signed saturating subtract vectors (unpredicated)

Signed saturating subtract all elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . This instruction is unpredicated.



SQSUB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - element2, esize, unsigned);

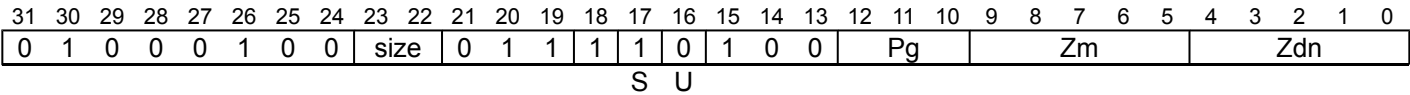
Z[d, VL] = result;
```



SQSUBR

Signed saturating subtraction reversed vectors (predicated)

Subtract active signed elements of the first source vector from corresponding signed elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Inactive elements in the destination vector register remain unmodified.



```
SQSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = SInt(Sat(element2 - element1, esize, unsigned));
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

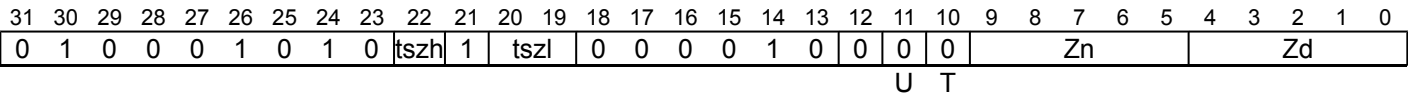
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQXTNB

Signed saturating extract narrow (bottom)

Saturate the signed integer value in each source element to half the original source element width, and place the results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero.



SQXTNB <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '001' esize = 16;
    when '010' esize = 32;
    when '100' esize = 64;
    otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	10	H
x	11	RESERVED
1	00	S
1	01	RESERVED
1	10	RESERVED

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	10	S
x	11	RESERVED
1	00	D
1	01	RESERVED
1	10	RESERVED

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result;
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    bits(halfesize) res = SignedSat(element1, halfesize);
    Elem[result, 2*e + 0, halfesize] = res;
    Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);

Z[d, VL] = result;
```

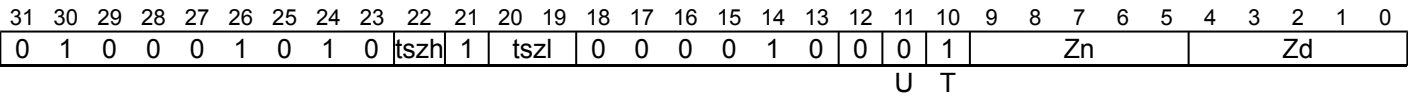
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQXTNT

Signed saturating extract narrow (top)

Saturate the signed integer value in each source element to half the original source element width, and place the results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged.



SQXTNT <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '001' esize = 16;
    when '010' esize = 32;
    when '100' esize = 64;
    otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	10	H
x	11	RESERVED
1	00	S
1	01	RESERVED
1	10	RESERVED

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	10	S
x	11	RESERVED
1	00	D
1	01	RESERVED
1	10	RESERVED

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(VL) operand1 = Z[n, VL];  
bits(VL) result = Z[d, VL];  
constant integer halfesize = esize DIV 2;  
  
for e = 0 to elements-1  
    integer element1 = SInt(Elem[operand1, e, esize]);  
    bits(halfesize) res = SignedSat(element1, halfesize);  
    Elem[result, 2*e + 1, halfesize] = res;  
  
Z[d, VL] = result;
```

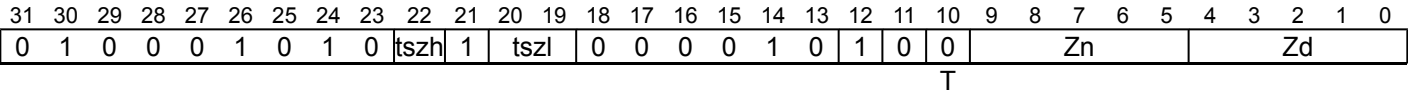
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQXTUNB

Signed saturating unsigned extract narrow (bottom)

Saturate the signed integer value in each source element to an unsigned integer value that is half the original source element width, and place the results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero.



SQXTUNB <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '001' esize = 16;
    when '010' esize = 32;
    when '100' esize = 64;
    otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	10	H
x	11	RESERVED
1	00	S
1	01	RESERVED
1	10	RESERVED

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	10	S
x	11	RESERVED
1	00	D
1	01	RESERVED
1	10	RESERVED

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(VL) operand1 = Z[n, VL];  
bits(VL) result;  
constant integer halfesize = esize DIV 2;  
  
for e = 0 to elements-1  
    integer element1 = SInt(Elem[operand1, e, esize]);  
    bits(halfesize) res = UnsignedSat(element1, halfesize);  
    Elem[result, 2*e + 0, halfesize] = res;  
    Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);  
  
Z[d, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

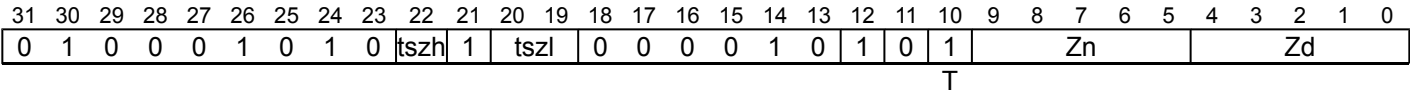
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SQXTUNT

Signed saturating unsigned extract narrow (top)

Saturate the signed integer value in each source element to an unsigned integer value that is half the original source element width, and place the results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged.



SQXTUNT <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '001' esize = 16;
    when '010' esize = 32;
    when '100' esize = 64;
    otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	10	H
x	11	RESERVED
1	00	S
1	01	RESERVED
1	10	RESERVED

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	10	S
x	11	RESERVED
1	00	D
1	01	RESERVED
1	10	RESERVED

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(VL) operand1 = Z[n, VL];  
bits(VL) result = Z[d, VL];  
constant integer halfesize = esize DIV 2;  
  
for e = 0 to elements-1  
    integer element1 = SInt(Elem[operand1, e, esize]);  
    bits(halfesize) res = UnsignedSat(element1, halfesize);  
    Elem[result, 2*e + 1, halfesize] = res;  
  
Z[d, VL] = result;
```

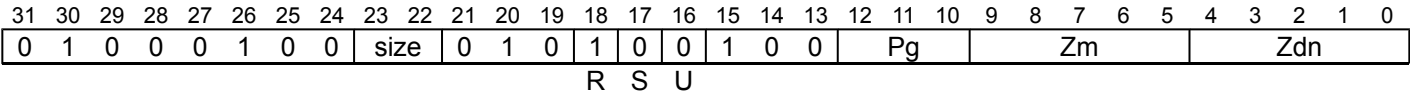
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SRHADD

Signed rounding halving addition

Add active signed elements of the first source vector to corresponding signed elements of the second source vector, shift right one bit, and destructively place the rounded results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



```
SRHADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = (element1 + element2 + 1) >> 1;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SRI

Shift right and insert (immediate)

Shift each source vector element right by an immediate value, and insert the result into the corresponding vector element in the destination vector register, merging the shifted bits from each source element with existing bits in each destination vector element. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	tszh	0	tszl	imm3			1	1	1	1	0	0	Zn						Zd					

SRI <Zd>.<T>, <Zn>.<T>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(esize) element1 = Elem[result, e, esize];
    bits(esize) element2 = Elem[operand, e, esize];
    bits(esize) mask = LSR(Ones(esize), shift);
    bits(esize) shiftedval = LSR(element2, shift);
    Elem[result, e, esize] = (element1 AND (NOT mask)) OR shiftedval;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

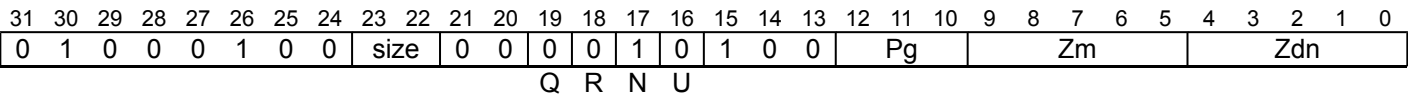
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SRSHL

Signed rounding shift left by vector (predicated)

Shift active signed elements of the first source vector by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.



SRSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    integer element = SInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer res;
    if shift >= 0 then
      res = element << shift;
    else
      shift = -shift;
      res = (element + (1 << (shift - 1))) >> shift;
    Elem[result, e, esize] = res<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

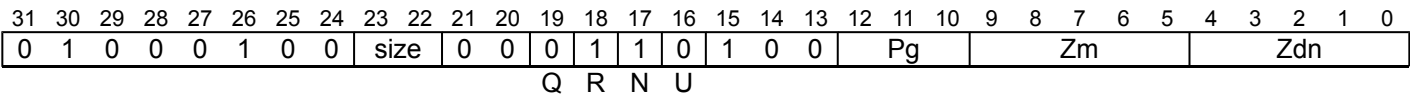
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SRSHLR

Signed rounding shift left reversed vectors (predicated)

Shift active signed elements of the second source vector by corresponding elements of the first source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.



```
SRSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand2 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    integer element = SInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer res;
    if shift >= 0 then
      res = element << shift;
    else
      shift = -shift;
      res = (element + (1 << (shift - 1))) >> shift;
    Elem[result, e, esize] = res<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand2, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

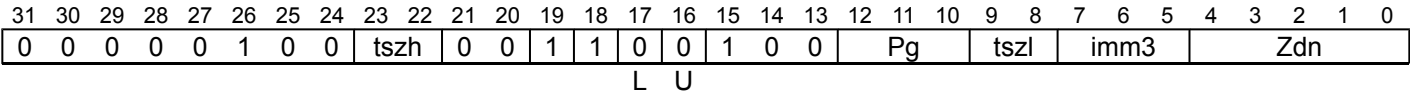
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SRSHR

Signed rounding shift right by immediate

Shift right by immediate each active signed element of the source vector, and destructively place the rounded results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.



SRSHR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

- <Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- <T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = (element1 + (1 << (shift - 1))) >> shift;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

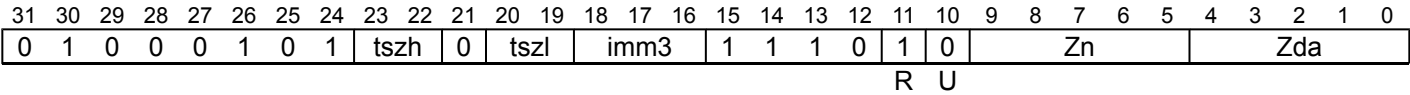
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SRSRA

Signed rounding shift right and accumulate (immediate)

Shift right by immediate each signed element of the source vector, preserving the sign bit, and add the rounded intermediate result destructively to the corresponding elements of the addend vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



```
SRSRA <Zda>.<T>, <Zn>.<T>, #<const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer da = UInt(Zda);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element = (SInt(Elem[operand1, e, esize]) + (1 << (shift - 1))) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

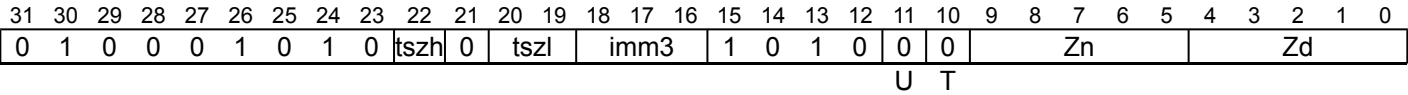
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



SSHLLB

Signed shift left long by immediate (bottom)

Shift left by immediate each even-numbered signed element of the source vector, and place the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.



```
SSHLLB <Zd>.<T>, <Zn>.<Tb>, #<const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand, 2*e + 0, esize];
    integer shifted_value = SInt(element) << shift;
    Elem[result, e, 2*esize] = shifted_value<2*esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

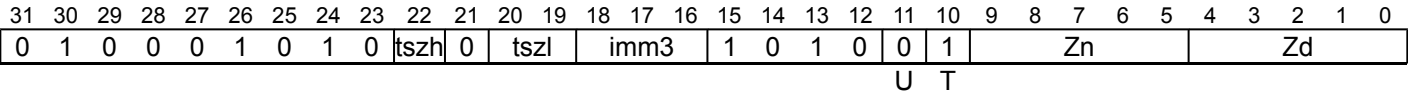
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SSHLLT

Signed shift left long by immediate (top)

Shift left by immediate each odd-numbered signed element of the source vector, and place the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.



```
SSHLLT <Zd>.<T>, <Zn>.<Tb>, #<const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand, 2*e + 1, esize];
    integer shifted_value = SInt(element) << shift;
    Elem[result, e, 2*esize] = shifted_value<2*esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

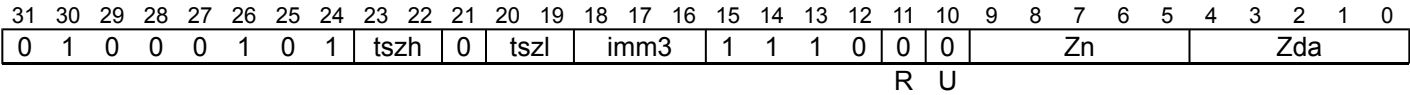
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSRA

Signed shift right and accumulate (immediate)

Shift right by immediate each signed element of the source vector, preserving the sign bit, and add the truncated intermediate result destructively to the corresponding elements of the addend vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



```
SSRA <Zda>.<T>, <Zn>.<T>, #<const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer da = UInt(Zda);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element = SInt(Elem[operand1, e, esize]) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

Z[da, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

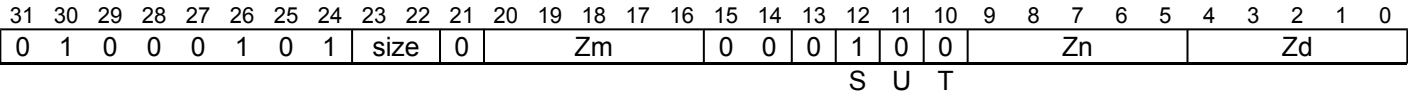
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSUBLB

Signed subtract long (bottom)

Subtract the even-numbered signed elements of the second source vector from the corresponding signed elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



```
SSUBLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 0;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

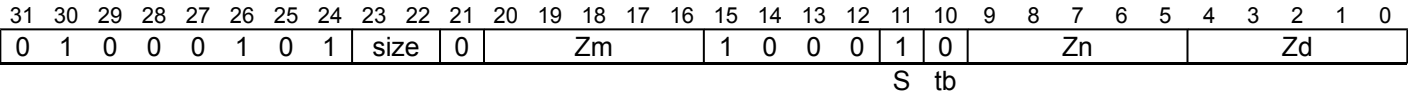
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSUBLBT

Signed subtract long (bottom - top)

Subtract the odd-numbered signed elements of the second source vector from the even-numbered signed elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



```
SSUBLBT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

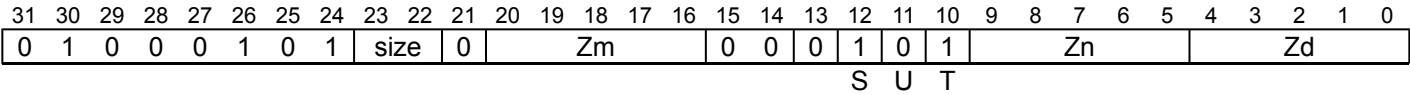
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SSUBLT

Signed subtract long (top)

Subtract the odd-numbered signed elements of the second source vector from the corresponding signed elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



```
SSUBLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

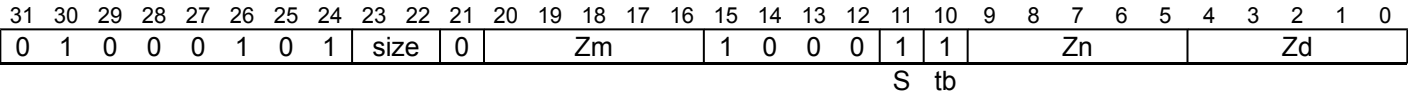
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSUBLTB

Signed subtract long (top - bottom)

Subtract the even-numbered signed elements of the second source vector from the odd-numbered signed elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



```
SSUBLTB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 0;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

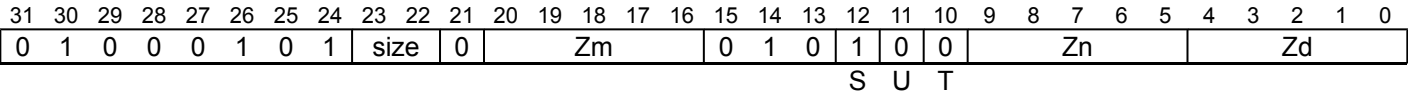
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSUBWB

Signed subtract wide (bottom)

Subtract the even-numbered signed elements of the second source vector from the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.



```
SSUBWB <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    Elem[result, e, esize] = (element1 - element2)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

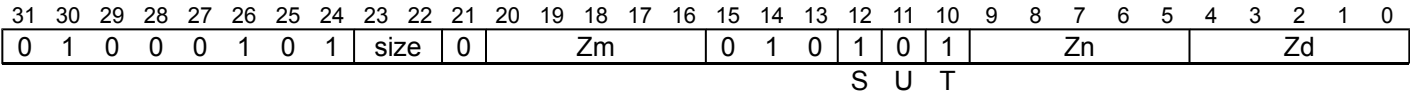
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSUBWT

Signed subtract wide (top)

Subtract the even-numbered signed elements of the second source vector from the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.



SSUBWT <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    Elem[result, e, esize] = (element1 - element2)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



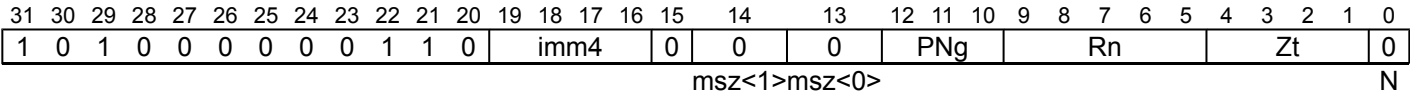
# ST1B (scalar plus immediate, consecutive registers)

Contiguous store of bytes from multiple consecutive vectors (immediate index)

Contiguous store of bytes from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.  
Inactive elements are not written to memory.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

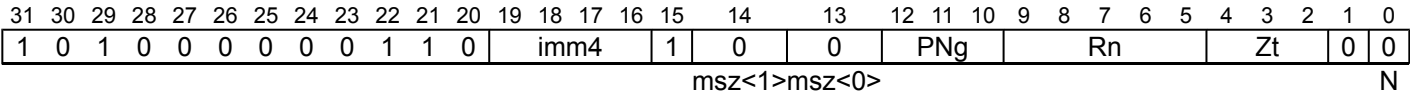
## Two registers (FEAT\_SVE2p1)



ST1B { <Zt1>.B-<Zt2>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 8;
integer offset = SInt(imm4);
```

## Four registers (FEAT\_SVE2p1)



ST1B { <Zt1>.B-<Zt4>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 8;
integer offset = SInt(imm4);
```

## Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.  
For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

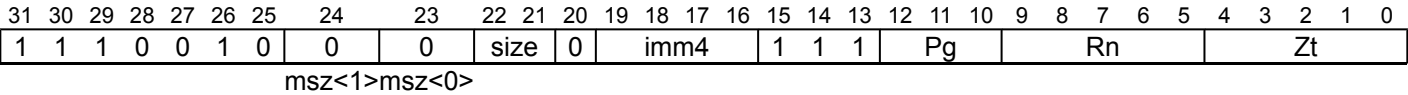
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST1B (scalar plus immediate, single register)

Contiguous store bytes from vector (immediate index)

Contiguous store of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.



```
ST1B { <Zt>.<T> }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8 << UInt(size);
constant integer msize = 8;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST1B (scalar plus scalar, consecutive registers)

Contiguous store of bytes from multiple consecutive vectors (scalar index)

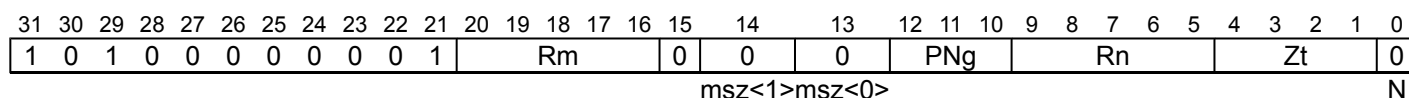
Contiguous store of bytes from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

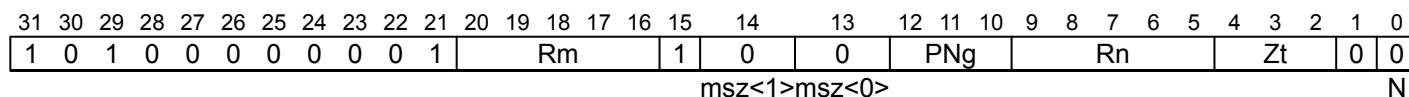


**ST1B** { <Zt1>.B-<Zt2>.B }, <PNg>, [<Xn|SP>, <Xm>]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 8;
```

### Four registers

(FEAT\_SVE2p1)



**ST1B** { <Zt1>.B-<Zt4>.B }, <PNg>, [<Xn|SP>, <Xm>]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 8;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.
- For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST1B (scalar plus scalar, single register)

Contiguous store bytes from vector (scalar index)

Contiguous store of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	0	size		Rm					0	1	0	Pg			Rn				Zt					

ST1B { <Zt>.<T> }, <Pg>, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8 << UInt(size);
constant integer msize = 8;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## ST1B (scalar plus vector)

Scatter store bytes from a vector (vector index)

Scatter store of bytes from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 3 classes: [32-bit unpacked unscaled offset](#) , [32-bit unscaled offset](#) and [64-bit unscaled offset](#)

### 32-bit unpacked unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	0	0	0	Zm			1	xs	0	Pg			Rn			Zt								

msz<1>msz<0>

**ST1B** { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
constant integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 32-bit unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	0	1	0	Zm			1	xs	0	Pg			Rn			Zt								

msz<1>msz<0>

**ST1B** { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
constant integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 64-bit unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	0	0	0	Zm				1	0	1	Pg			Rn				Zt						

msz<1>msz<0>

ST1B { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

## Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
  
- <mod> Is the index extend and shift specifier, encoded in "xs":

xs	<mod>
0	UXTW
1	SXTW

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.



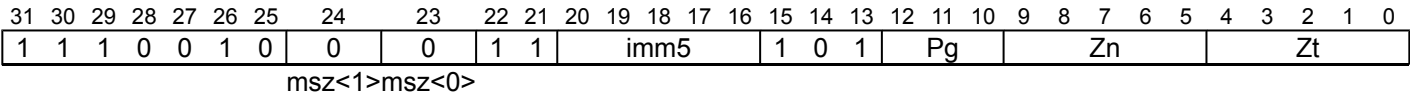
## ST1B (vector plus immediate)

Scatter store bytes from a vector (immediate index)

Scatter store of bytes from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements are not written to memory. This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

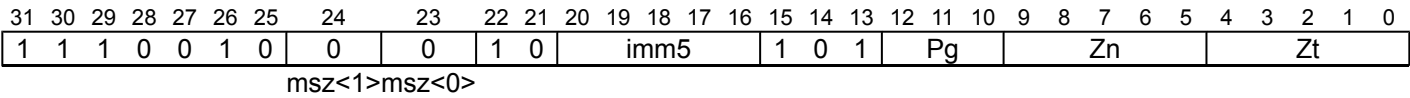
### 32-bit element



ST1B { <Zt>.S }, <Pg>, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
integer offset = UInt(imm5);
```

### 64-bit element



ST1B { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offset = UInt(imm5);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST1D (scalar plus immediate, consecutive registers)

Contiguous store of doublewords from multiple consecutive vectors (immediate index)

Contiguous store of doublewords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	1	1	0	imm4			0	1	1	PNg			Rn				Zt			0			
																	msz<1>msz<0>										N				

ST1D { <Zt1>.D-<Zt2>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	1	1	0	imm4			1	1	1	PNg			Rn				Zt			0	0		
																	msz<1>msz<0>										N				

ST1D { <Zt1>.D-<Zt4>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.
- For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

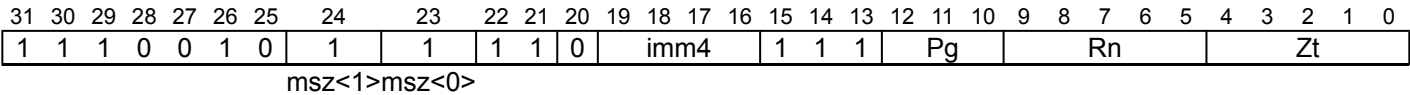
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# ST1D (scalar plus immediate, single register)

Contiguous store doublewords from vector (immediate index)

Contiguous store of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory. It has encodings from 2 classes: [SVE](#) and [SVE2](#)

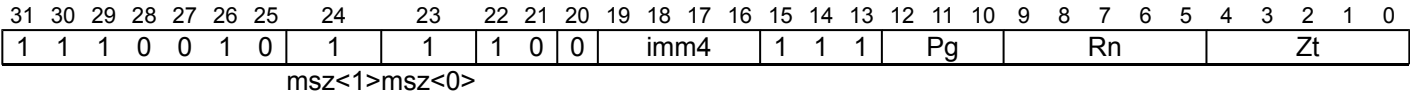
## SVE



ST1D { <Zt>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offset = SInt(imm4);
```

## SVE2 (FEAT\_SVE2p1)



ST1D { <Zt>.Q }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE2p1() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 128;
constant integer msize = 64;
integer offset = SInt(imm4);
```

## Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.



## Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

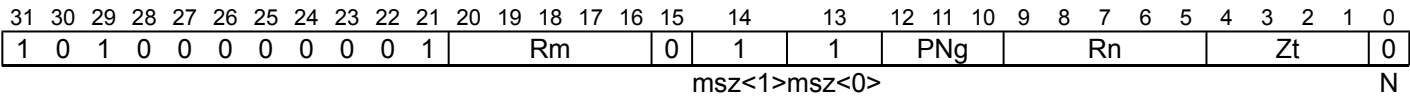
# ST1D (scalar plus scalar, consecutive registers)

Contiguous store of doublewords from multiple consecutive vectors (scalar index)

Contiguous store of doublewords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.  
Inactive elements are not written to memory.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

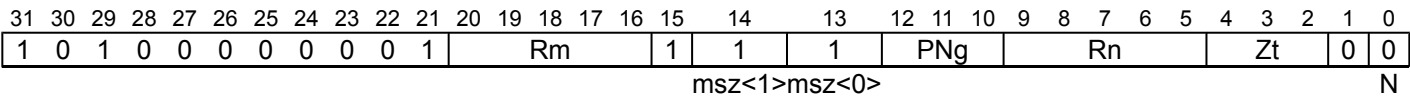
## Two registers (FEAT\_SVE2p1)



ST1D { <Zt1>.D-<Zt2>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 64;
```

## Four registers (FEAT\_SVE2p1)



ST1D { <Zt1>.D-<Zt4>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 64;
```

## Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.  
For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST1D (scalar plus scalar, single register)

Contiguous store doublewords from vector (scalar index)

Contiguous store of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory. It has encodings from 2 classes: [SVE](#) and [SVE2](#)

### SVE

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	1	1	1	1	Rm				0	1	0	Pg			Rn				Zt						

ST1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
```

### SVE2 (FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	1	1	1	0	Rm				0	1	0	Pg			Rn				Zt						

ST1D { <Zt>.Q }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE2p1() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 128;
constant integer msize = 64;
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST1D (scalar plus vector)

Scatter store doublewords from a vector (vector index)

Scatter store of doublewords from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 4 classes: [32-bit unpacked scaled offset](#) , [32-bit unpacked unscaled offset](#) , [64-bit scaled offset](#) and [64-bit unscaled offset](#)

### 32-bit unpacked scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	1	1	0	1	Zm				1	xs	0	Pg			Rn				Zt						
msz<1>msz<0>																															

**ST1D** { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #3]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
constant integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 3;
```

### 32-bit unpacked unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	1	1	0	0	Zm				1	xs	0	Pg			Rn				Zt						
msz<1>msz<0>																															

**ST1D** { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
constant integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

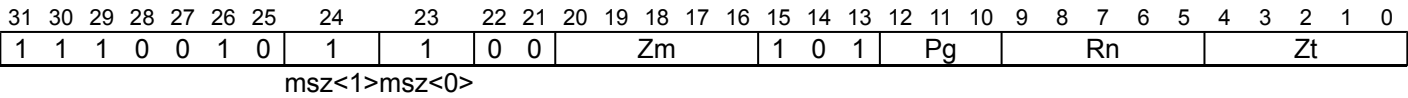
### 64-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	1	1	0	1	Zm				1	0	1	Pg			Rn				Zt						
msz<1>msz<0>																															

ST1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, LSL #3]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 3;
```

64-bit unscaled offset



ST1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
- <mod> Is the index extend and shift specifier, encoded in "xs":

xs	<mod>
0	UXTW
1	SXTW

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

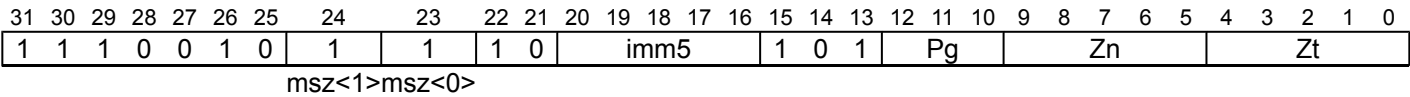


# ST1D (vector plus immediate)

Scatter store doublewords from a vector (immediate index)

Scatter store of doublewords from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.



ST1D { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offset = UInt(imm5);
```

## Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.



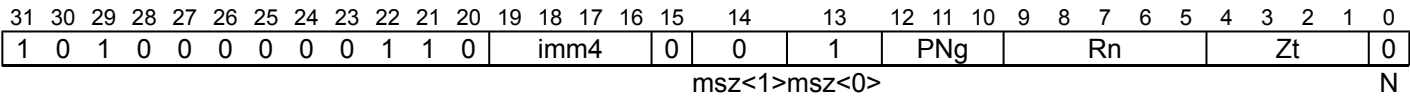
# ST1H (scalar plus immediate, consecutive registers)

Contiguous store of halfwords from multiple consecutive vectors (immediate index)

Contiguous store of halfwords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.  
Inactive elements are not written to memory.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

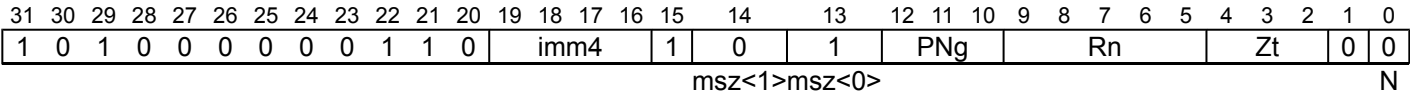
## Two registers (FEAT\_SVE2p1)



ST1H { <Zt1>.H-<Zt2>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 16;
integer offset = SInt(imm4);
```

## Four registers (FEAT\_SVE2p1)



ST1H { <Zt1>.H-<Zt4>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 16;
integer offset = SInt(imm4);
```

## Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.  
For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

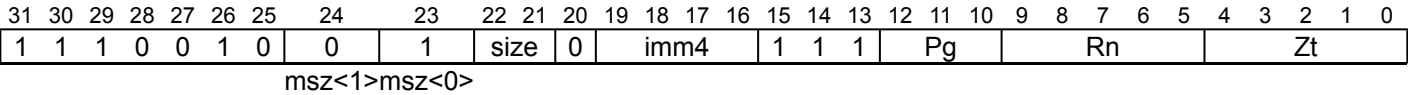
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST1H (scalar plus immediate, single register)

Contiguous store halfwords from vector (immediate index)

Contiguous store of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.



```
ST1H { <Zt>.<T> }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8 << UInt(size);
constant integer msize = 16;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

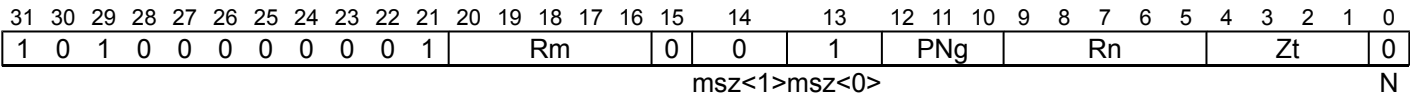
# ST1H (scalar plus scalar, consecutive registers)

Contiguous store of halfwords from multiple consecutive vectors (scalar index)

Contiguous store of halfwords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.  
Inactive elements are not written to memory.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

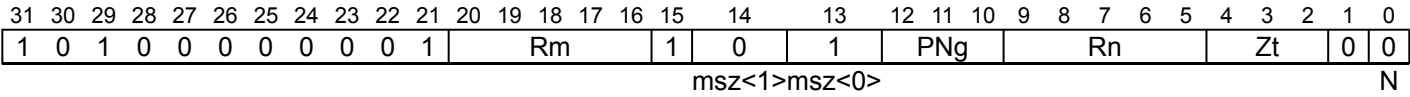
## Two registers (FEAT\_SVE2p1)



ST1H { <Zt1>.H-<Zt2>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 16;
```

## Four registers (FEAT\_SVE2p1)



ST1H { <Zt1>.H-<Zt4>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 16;
```

## Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.  
For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



ST1H (scalar plus scalar, single register)

Contiguous store halfwords from vector (scalar index)

Contiguous store of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	1	size			Rm				0	1	0	Pg			Rn				Zt					

ST1H { <Zt>.<T> }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8 << UInt(size);
constant integer msize = 16;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST1H (scalar plus vector)

Scatter store halfwords from a vector (vector index)

Scatter store of halfwords from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 6 classes: [32-bit scaled offset](#) , [32-bit unpacked scaled offset](#) , [32-bit unpacked unscaled offset](#) , [32-bit unscaled offset](#) , [64-bit scaled offset](#) and [64-bit unscaled offset](#)

### 32-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	1	1	1	Zm			1	xs	0	Pg			Rn			Zt								
msz<1>msz<0>																															

**ST1H** { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
constant integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

### 32-bit unpacked scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	1	0	1	Zm			1	xs	0	Pg			Rn			Zt								
msz<1>msz<0>																															

**ST1H** { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

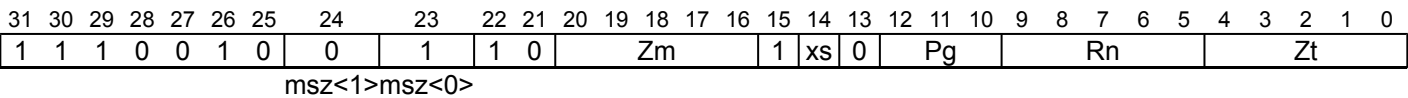
### 32-bit unpacked unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	1	0	0	Zm			1	xs	0	Pg			Rn			Zt								
msz<1>msz<0>																															

**ST1H** { **<Zt>.D** }, **<Pg>**, [**<Xn|SP>**, **<Zm>.D**, **<mod>**]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

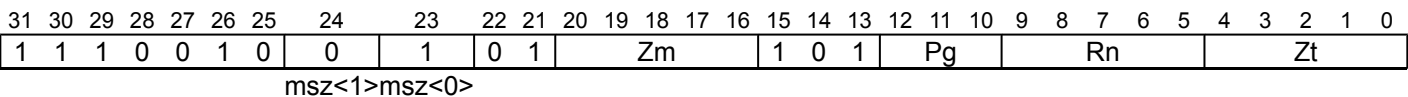
### 32-bit unscaled offset



**ST1H** { **<Zt>.S** }, **<Pg>**, [**<Xn|SP>**, **<Zm>.S**, **<mod>**]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
constant integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

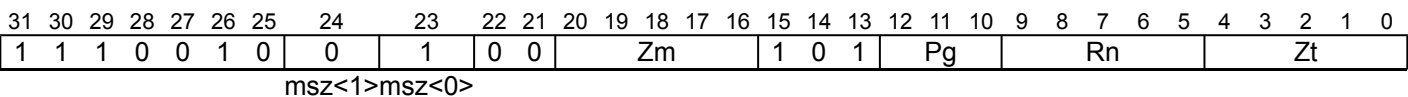
### 64-bit scaled offset



**ST1H** { **<Zt>.D** }, **<Pg>**, [**<Xn|SP>**, **<Zm>.D**, **LSL #1**]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 1;
```

### 64-bit unscaled offset



ST1H { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

## Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
- <mod> Is the index extend and shift specifier, encoded in "xs":

xs	<mod>
0	UXTW
1	SXTW

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.



## ST1H (vector plus immediate)

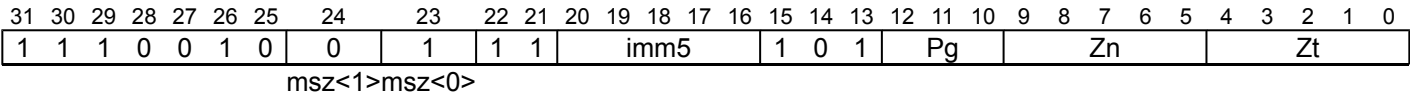
Scatter store halfwords from a vector (immediate index)

Scatter store of halfwords from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

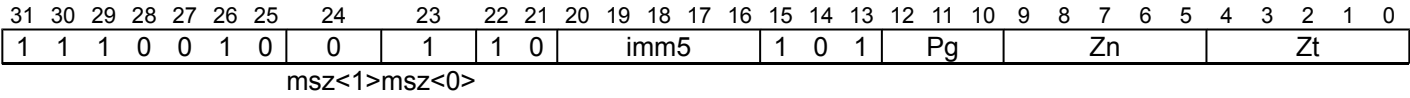
### 32-bit element



ST1H { <Zt>.S }, <Pg>, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offset = UInt(imm5);
```

### 64-bit element



ST1H { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offset = UInt(imm5);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# ST1Q

Scatter store quadwords

Scatter store of quadwords from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory. This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

## SVE2 (FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	0	0	1	Rm				0	0	1	Pg			Zn				Zt						

ST1Q { <Zt>.Q }, <Pg>, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2p1() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

### Operation

```
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
CheckNonStreamingSVEEnabled();
constant integer elements = VL DIV 128;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) src;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, 128) then
    base = Z[n, VL];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, 128) then
        bits(64) addr = Elem[base, 2*e, 64] + offset;
        Mem[addr, 16, accdesc] = Elem[src, e, 128];
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.



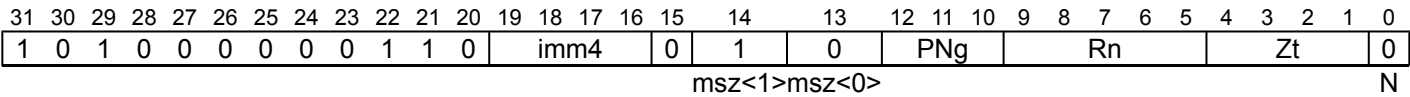
# ST1W (scalar plus immediate, consecutive registers)

Contiguous store of words from multiple consecutive vectors (immediate index)

Contiguous store of words from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.  
Inactive elements are not written to memory.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

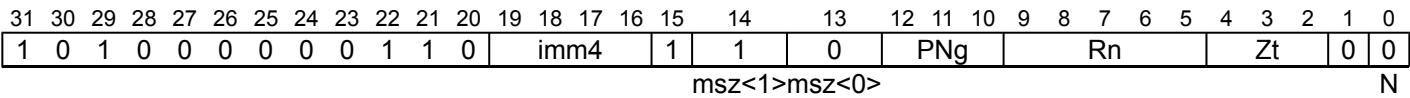
## Two registers (FEAT\_SVE2p1)



ST1W { <Zt1>.S-<Zt2>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 32;
integer offset = SInt(imm4);
```

## Four registers (FEAT\_SVE2p1)



ST1W { <Zt1>.S-<Zt4>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 32;
integer offset = SInt(imm4);
```

## Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.  
For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

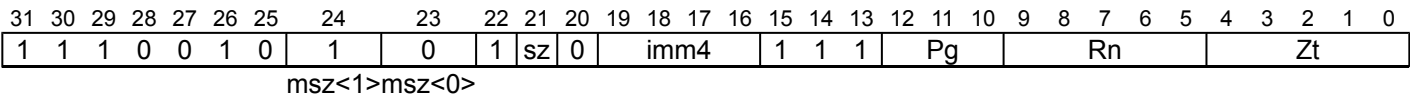
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST1W (scalar plus immediate, single register)

Contiguous store words from vector (immediate index)

Contiguous store of words from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory. It has encodings from 2 classes: [SVE](#) and [SVE2](#)

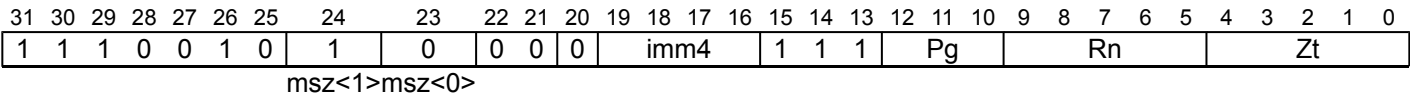
SVE



ST1W { <Zt>.<T> }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32 << UInt(sz);
constant integer msize = 32;
integer offset = SInt(imm4);
```

SVE2  
(FEAT\_SVE2p1)



ST1W { <Zt>.Q }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE2p1() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 128;
constant integer msize = 32;
integer offset = SInt(imm4);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

## Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

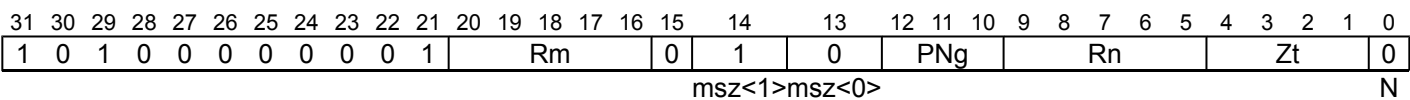
# ST1W (scalar plus scalar, consecutive registers)

Contiguous store of words from multiple consecutive vectors (scalar index)

Contiguous store of words from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.  
Inactive elements are not written to memory.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

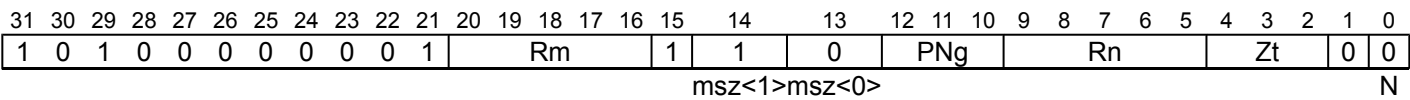
## Two registers (FEAT\_SVE2p1)



ST1W { <Zt1>.S-<Zt2>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 32;
```

## Four registers (FEAT\_SVE2p1)



ST1W { <Zt1>.S-<Zt4>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 32;
```

## Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.  
For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



ST1W (scalar plus scalar, single register)

Contiguous store words from vector (scalar index)

Contiguous store of words from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory. It has encodings from 2 classes: [SVE](#) and [SVE2](#)

SVE

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	1	0	1	sz	Rm				0	1	0	Pg			Rn				Zt						

ST1W { <Zt>.<T> }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32 << UInt(sz);
constant integer msize = 32;
```

SVE2  
(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	1	0	0	0	Rm				0	1	0	Pg			Rn				Zt						

ST1W { <Zt>.Q }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE2p1() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 128;
constant integer msize = 32;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST1W (scalar plus vector)

Scatter store words from a vector (vector index)

Scatter store of words from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 6 classes: [32-bit scaled offset](#) , [32-bit unpacked scaled offset](#) , [32-bit unpacked unscaled offset](#) , [32-bit unscaled offset](#) , [64-bit scaled offset](#) and [64-bit unscaled offset](#)

### 32-bit scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	1	0	1	1	Zm				1	xs	0	Pg				Rn				Zt					
msz<1>msz<0>																															

**ST1W** { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
constant integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

### 32-bit unpacked scaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	1	0	0	1	Zm				1	xs	0	Pg				Rn				Zt					
msz<1>msz<0>																															

**ST1W** { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

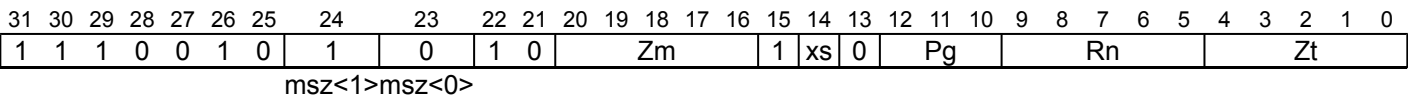
### 32-bit unpacked unscaled offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	1	0	0	0	Zm				1	xs	0	Pg				Rn				Zt					
msz<1>msz<0>																															

**ST1W** { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

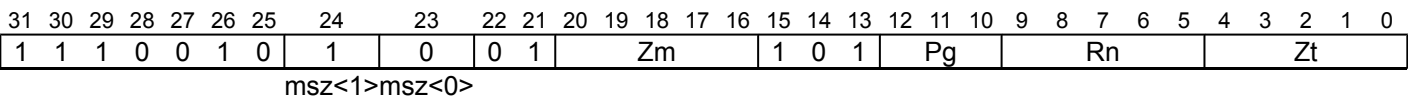
### 32-bit unscaled offset



**ST1W** { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
constant integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

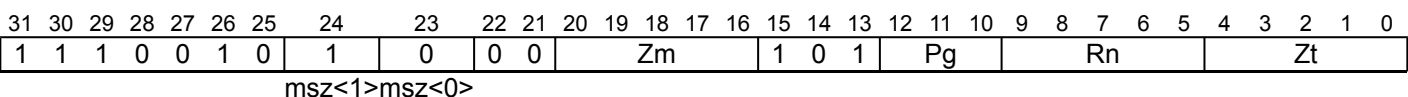
### 64-bit scaled offset



**ST1W** { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, LSL #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 2;
```

### 64-bit unscaled offset



ST1W { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

## Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
- <mod> Is the index extend and shift specifier, encoded in "xs":

xs	<mod>
0	UXTW
1	SXTW

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.



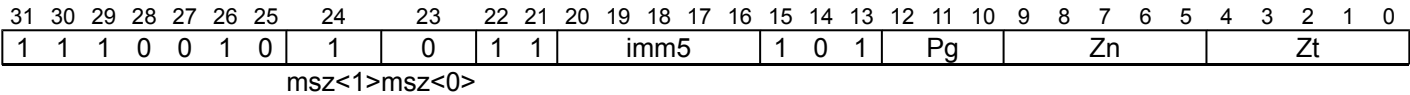
ST1W (vector plus immediate)

Scatter store words from a vector (immediate index)

Scatter store of words from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements are not written to memory.  
This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit element](#) and [64-bit element](#)

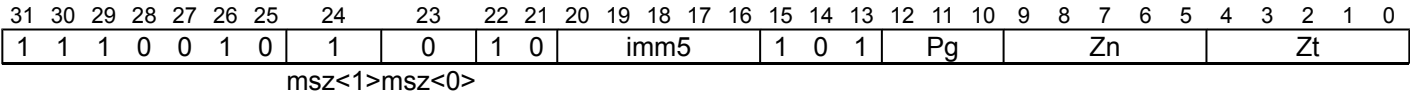
32-bit element



```
ST1W { <Zt>.S }, <Pg>, [<Zn>.S{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
integer offset = UInt(imm5);
```

64-bit element



```
ST1W { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offset = UInt(imm5);
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

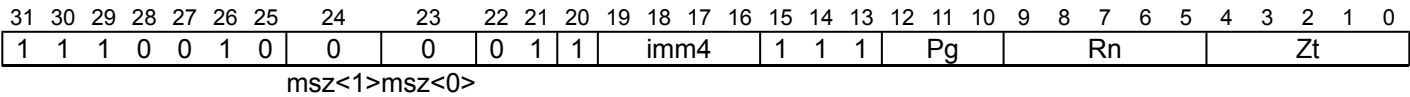


ST2B (scalar plus immediate)

Contiguous store two-byte structures from two vectors (immediate index)

Contiguous store two-byte structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.



```
ST2B { <Zt1>.B, <Zt2>.B }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
constant integer nreg = 2;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

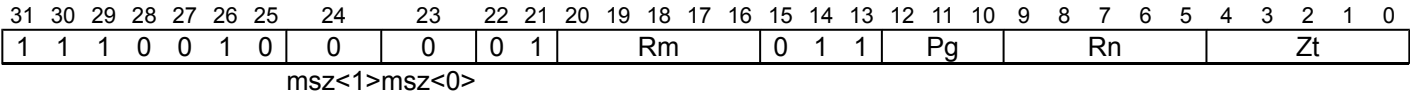
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST2B (scalar plus scalar)

Contiguous store two-byte structures from two vectors (scalar index)

Contiguous store two-byte structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.



ST2B { <Zt1>.B, <Zt2>.B }, <Pg>, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer nreg = 2;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

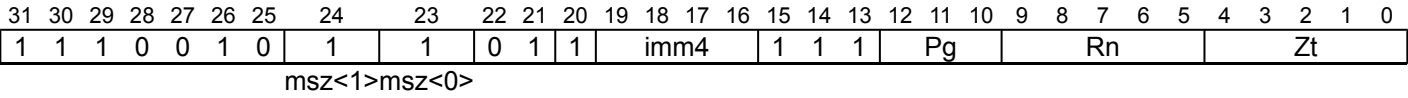
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST2D (scalar plus immediate)

Contiguous store two-doubleword structures from two vectors (immediate index)

Contiguous store two-doubleword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.



```
ST2D { <Zt1>.D, <Zt2>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
constant integer nreg = 2;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

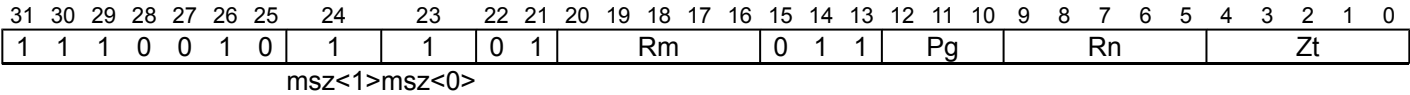
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST2D (scalar plus scalar)

Contiguous store two-doubleword structures from two vectors (scalar index)

Contiguous store two-doubleword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.



ST2D { <Zt1>.D, <Zt2>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer nreg = 2;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

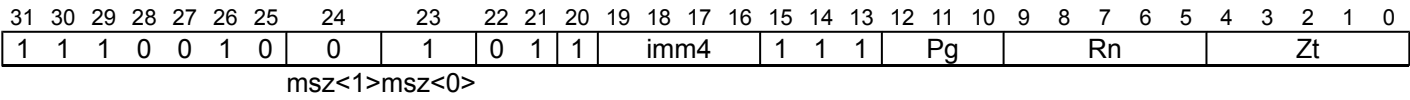


ST2H (scalar plus immediate)

Contiguous store two-halfword structures from two vectors (immediate index)

Contiguous store two-halfword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.



```
ST2H { <Zt1>.H, <Zt2>.H }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
constant integer nreg = 2;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

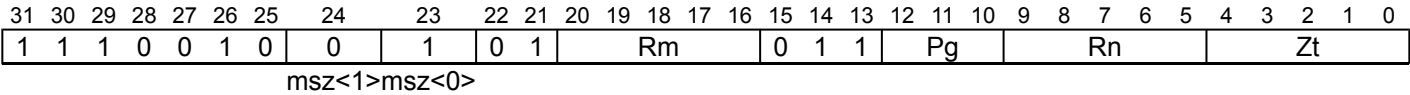
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST2H (scalar plus scalar)

Contiguous store two-halfword structures from two vectors (scalar index)

Contiguous store two-halfword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.



ST2H { <Zt1>.H, <Zt2>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer nreg = 2;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

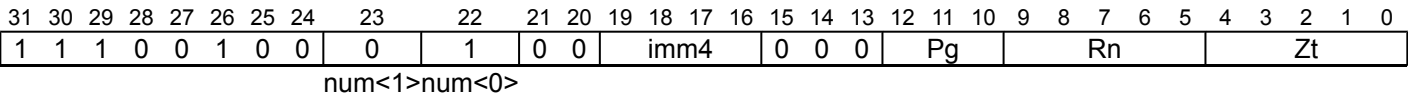
## ST2Q (scalar plus immediate)

Contiguous store two-quadword structures from two vectors (immediate index)

Contiguous store two-quadword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive quadwords in memory which make up each structure. Inactive structures are not written to memory.

### SVE2 (FEAT\_SVE2p1)



ST2Q { <Zt1>.Q, <Zt2>.Q }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 128;
integer offset = SInt(imm4);
constant integer nreg = 2;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

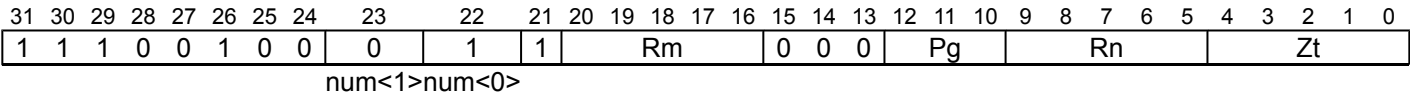
## ST2Q (scalar plus scalar)

Contiguous store two-quadword structures from two vectors (scalar index)

Contiguous store two-quadword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive quadwords in memory which make up each structure. Inactive structures are not written to memory.

### SVE2 (FEAT\_SVE2p1)



ST2Q { <Zt1>.Q, <Zt2>.Q }, <Pg>, [<Xn|SP>, <Xm>, LSL #4]

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 128;
constant integer nreg = 2;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

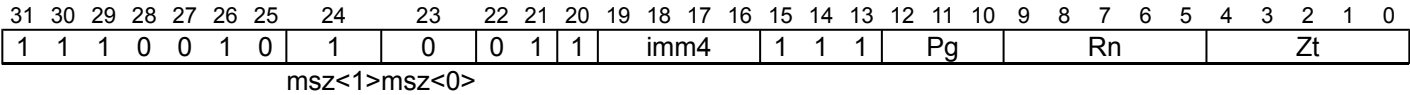


## ST2W (scalar plus immediate)

Contiguous store two-word structures from two vectors (immediate index)

Contiguous store two-word structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive structures are not written to memory.



ST2W { <Zt1>.S, <Zt2>.S }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
constant integer nreg = 2;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

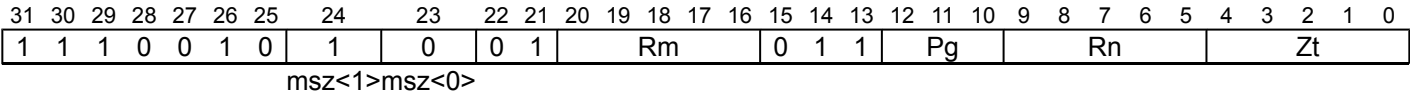
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST2W (scalar plus scalar)

Contiguous store two-word structures from two vectors (scalar index)

Contiguous store two-word structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive structures are not written to memory.



ST2W { <Zt1>.S, <Zt2>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer nreg = 2;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

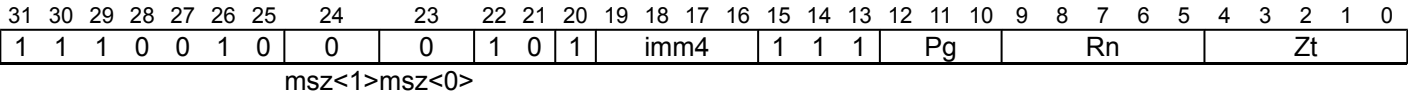
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST3B (scalar plus immediate)

Contiguous store three-byte structures from three vectors (immediate index)

Contiguous store three-byte structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.



```
ST3B { <Zt1>.B, <Zt2>.B, <Zt3>.B }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

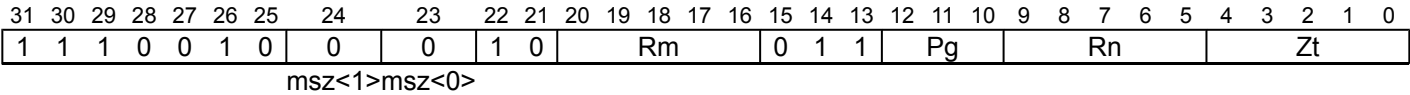
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST3B (scalar plus scalar)

Contiguous store three-byte structures from three vectors (scalar index)

Contiguous store three-byte structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.



ST3B { <Zt1>.B, <Zt2>.B, <Zt3>.B }, <Pg>, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer nreg = 3;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

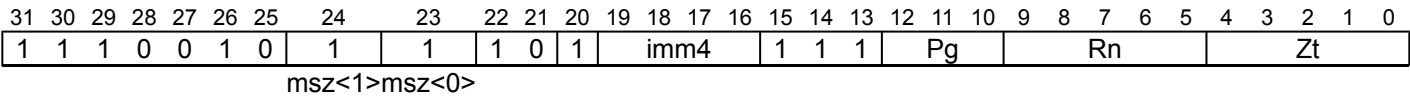
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



ST3D (scalar plus immediate)

Contiguous store three-doubleword structures from three vectors (immediate index)

Contiguous store three-doubleword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication,  
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.



```
ST3D { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

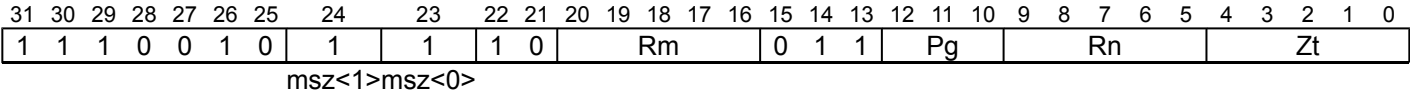
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST3D (scalar plus scalar)

Contiguous store three-doubleword structures from three vectors (scalar index)

Contiguous store three-doubleword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.



ST3D { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer nreg = 3;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

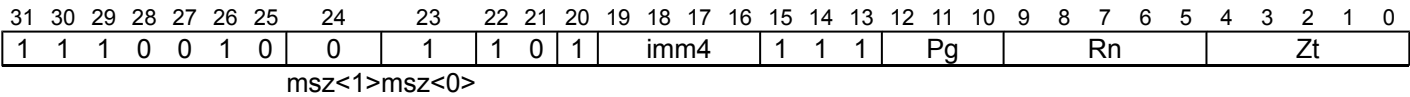
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST3H (scalar plus immediate)

Contiguous store three-halfword structures from three vectors (immediate index)

Contiguous store three-halfword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication, Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.



ST3H { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

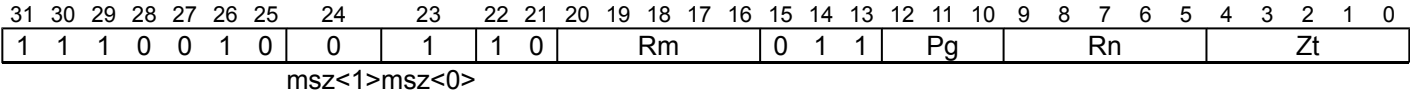
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST3H (scalar plus scalar)

Contiguous store three-halfword structures from three vectors (scalar index)

Contiguous store three-halfword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.



**ST3H** { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer nreg = 3;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

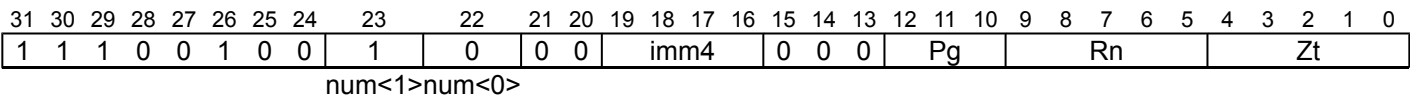


## ST3Q (scalar plus immediate)

Contiguous store three-quadword structures from three vectors (immediate index)

Contiguous store three-quadword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication,  
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive quadwords in memory which make up each structure. Inactive structures are not written to memory.

### SVE2 (FEAT\_SVE2p1)



ST3Q { <Zt1>.Q, <Zt2>.Q, <Zt3>.Q }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 128;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

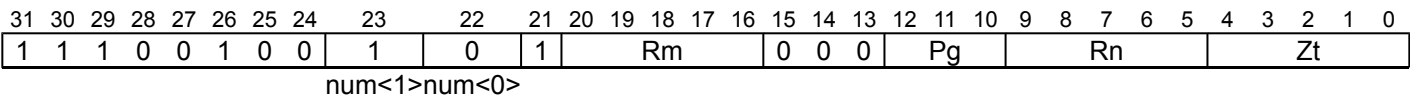
## ST3Q (scalar plus scalar)

Contiguous store three-quadword structures from three vectors (scalar index)

Contiguous store three-quadword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive quadwords in memory which make up each structure. Inactive structures are not written to memory.

### SVE2 (FEAT\_SVE2p1)



ST3Q { <Zt1>.Q, <Zt2>.Q, <Zt3>.Q }, <Pg>, [<Xn|SP>, <Xm>, LSL #4]

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 128;
constant integer nreg = 3;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

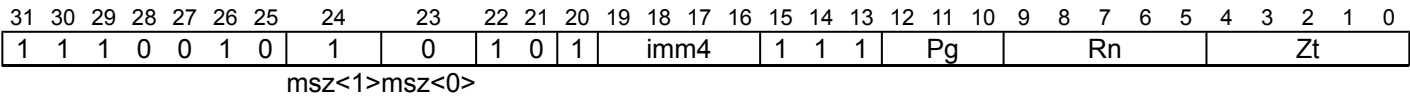
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST3W (scalar plus immediate)

Contiguous store three-word structures from three vectors (immediate index)

Contiguous store three-word structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive structures are not written to memory.



ST3W { <Zt1>.S, <Zt2>.S, <Zt3>.S }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

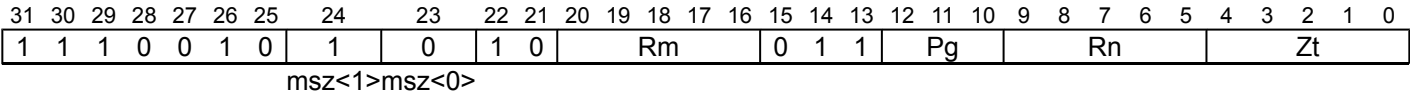
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST3W (scalar plus scalar)

Contiguous store three-word structures from three vectors (scalar index)

Contiguous store three-word structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive structures are not written to memory.



ST3W { <Zt1>.S, <Zt2>.S, <Zt3>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer nreg = 3;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

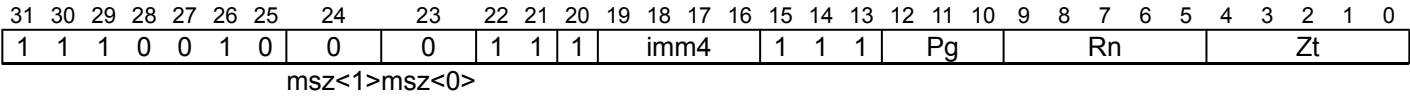


ST4B (scalar plus immediate)

Contiguous store four-byte structures from four vectors (immediate index)

Contiguous store four-byte structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.



```
ST4B { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
constant integer nreg = 4;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

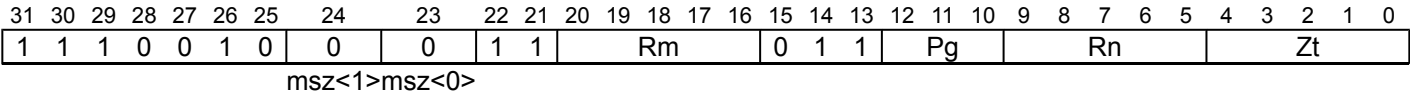
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST4B (scalar plus scalar)

Contiguous store four-byte structures from four vectors (scalar index)

Contiguous store four-byte structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.



ST4B { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <Pg>, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer nreg = 4;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

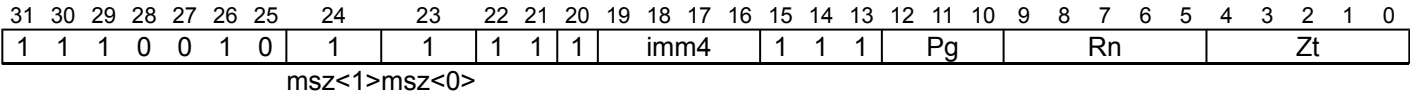
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST4D (scalar plus immediate)

Contiguous store four-doubleword structures from four vectors (immediate index)

Contiguous store four-doubleword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.



**ST4D** { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
constant integer nreg = 4;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

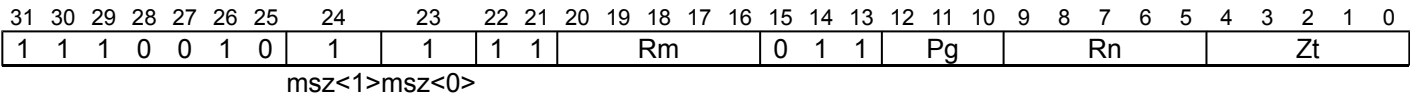
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST4D (scalar plus scalar)

Contiguous store four-doubleword structures from four vectors (scalar index)

Contiguous store four-doubleword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.



ST4D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer nreg = 4;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

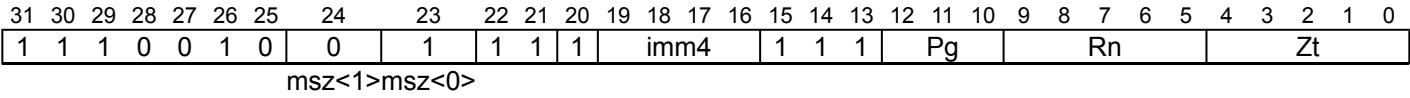


ST4H (scalar plus immediate)

Contiguous store four-halfword structures from four vectors (immediate index)

Contiguous store four-halfword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.



```
ST4H { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
constant integer nreg = 4;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

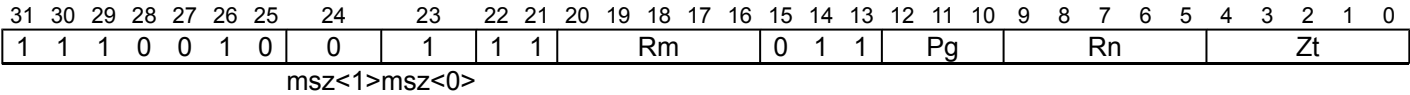
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST4H (scalar plus scalar)

Contiguous store four-halfword structures from four vectors (scalar index)

Contiguous store four-halfword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.



ST4H { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer nreg = 4;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

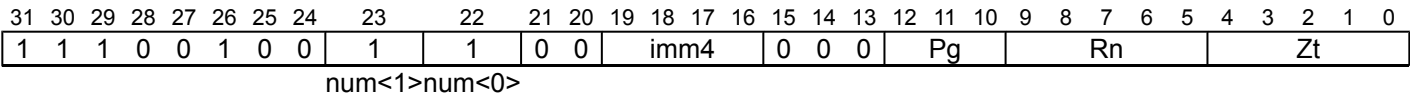
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST4Q (scalar plus immediate)

Contiguous store four-quadword structures from four vectors (immediate index)

Contiguous store four-quadword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication, Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive quadwords in memory which make up each structure. Inactive structures are not written to memory.

SVE2
(FEAT\_SVE2p1)



ST4Q { <Zt1>.Q, <Zt2>.Q, <Zt3>.Q, <Zt4>.Q }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 128;
integer offset = SInt(imm4);
constant integer nreg = 4;
```

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

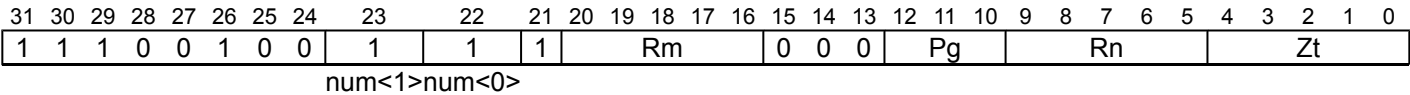
## ST4Q (scalar plus scalar)

Contiguous store four-quadword structures from four vectors (scalar index)

Contiguous store four-quadword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive quadwords in memory which make up each structure. Inactive structures are not written to memory.

### SVE2 (FEAT\_SVE2p1)



ST4Q { <Zt1>.Q, <Zt2>.Q, <Zt3>.Q, <Zt4>.Q }, <Pg>, [<Xn|SP>, <Xm>, LSL #4]

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 128;
constant integer nreg = 4;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

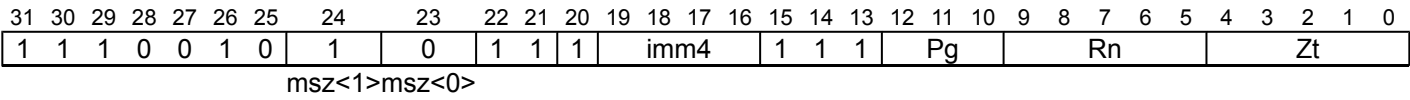


## ST4W (scalar plus immediate)

Contiguous store four-word structures from four vectors (immediate index)

Contiguous store four-word structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive structures are not written to memory.



ST4W { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
constant integer nreg = 4;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

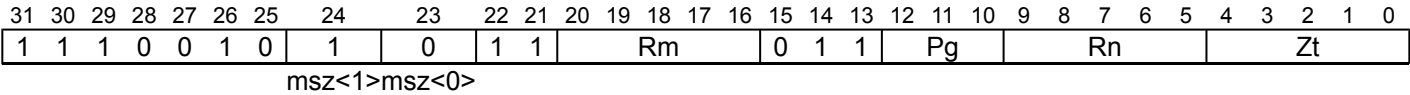
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST4W (scalar plus scalar)

Contiguous store four-word structures from four vectors (scalar index)

Contiguous store four-word structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive structures are not written to memory.



ST4W { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer nreg = 4;
```

### Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ActivePredicateElement(mask, e, esize) then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[values[r], e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1B (scalar plus immediate, consecutive registers)

Contiguous store non-temporal of bytes from multiple consecutive vectors (immediate index)

Contiguous store non-temporal of bytes from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	1	1	0	imm4				0	0	0	PNg				Rn				Zt				1
																msz<1>msz<0>												N			

```
STNT1B { <Zt1>.B-<Zt2>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 8;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	1	0	0	0	0	0	0	1	1	0	imm4				1	0	0	PNg				Rn				Zt				0	1
																msz<1>msz<0>														N		

```
STNT1B { <Zt1>.B-<Zt4>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 8;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.
- For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1B (scalar plus immediate, single register)

Contiguous store non-temporal bytes from vector (immediate index)

Contiguous store non-temporal of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	0	0	0	1	imm4				1	1	1	Pg			Rn				Zt					
msz<1>msz<0>																															

**STNT1B** { <Zt>.B }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g, PL];
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## STNT1B (scalar plus scalar, consecutive registers)

Contiguous store non-temporal of bytes from multiple consecutive vectors (scalar index)

Contiguous store non-temporal of bytes from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm				0	0	0	PNg	Rn				Zt				1				
																msz<1>msz<0>										N					

STNT1B { <Zt1>.B-<Zt2>.B }, <PNg>, [<Xn|SP>, <Xm>]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 8;
```

### Four registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm				1	0	0	PNg			Rn				Zt			0	1		
																msz<1>msz<0>														N	

STNT1B { <Zt1>.B-<Zt4>.B }, <PNg>, [<Xn|SP>, <Xm>]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 8;
```

### Assembler Symbols

<Zt1>	For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.  For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1B (scalar plus scalar, single register)

Contiguous store non-temporal bytes from vector (scalar index)

Contiguous store non-temporal of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	0	0	0	Rm				0	1	1	Pg			Rn				Zt						
msz<1>msz<0>																															

**STNT1B** { <Zt>.B }, <Pg>, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(VL) src;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STNT1B (vector plus scalar)

Scatter store non-temporal bytes

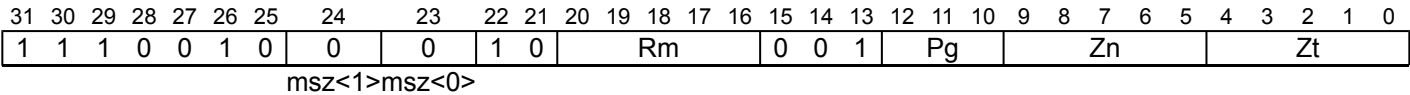
Scatter store non-temporal of bytes from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit unscaled offset](#) and [64-bit unscaled offset](#)

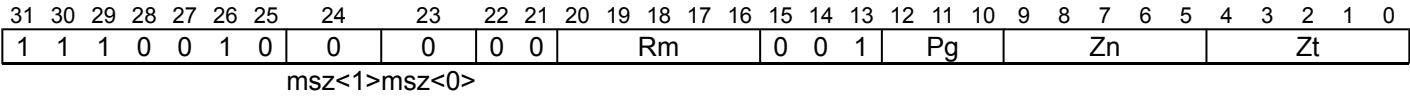
32-bit unscaled offset



STNT1B { <Zt>.S }, <Pg>, [<Zn>.S{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
```

64-bit unscaled offset



STNT1B { <Zt>.D }, <Pg>, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1D (scalar plus immediate, consecutive registers)

Contiguous store non-temporal of doublewords from multiple consecutive vectors (immediate index)

Contiguous store non-temporal of doublewords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

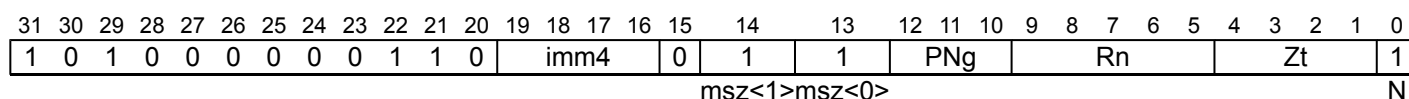
Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

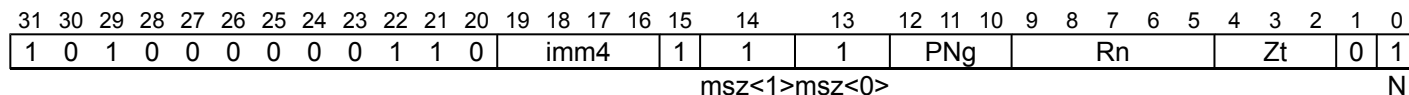


STNT1D { <Zt1>.D-<Zt2>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SVE2p1)



STNT1D { <Zt1>.D-<Zt4>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.
- For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## STNT1D (scalar plus immediate, single register)

Contiguous store non-temporal doublewords from vector (immediate index)

Contiguous store non-temporal of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	1	1	0	0	1	imm4				1	1	1	Pg			Rn			Zt						
msz<1>msz<0>																															

**STNT1D** { <Zt>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g, PL];
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1D (scalar plus scalar, consecutive registers)

Contiguous store non-temporal of doublewords from multiple consecutive vectors (scalar index)

Contiguous store non-temporal of doublewords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm				0	1	1	PNg	Rn				Zt				1				
																msz<1>msz<0>														N	

STNT1D { <Zt1>.D-<Zt2>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 64;
```

### Four registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm				1	1	1	PNg			Rn				Zt			0	1		
																msz<1>msz<0>															

STNT1D { <Zt1>.D-<Zt4>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 64;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.
- For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1D (scalar plus scalar, single register)

Contiguous store non-temporal doublewords from vector (scalar index)

Contiguous store non-temporal of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	1	1	0	0	Rm				0	1	1	Pg			Rn				Zt						
msz<1>msz<0>																															

```
STNT1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(VL) src;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

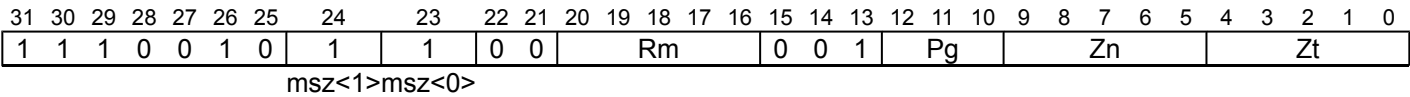
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1D (vector plus scalar)

Scatter store non-temporal doublewords

Scatter store non-temporal of doublewords from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory. A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.



STNT1D { <Zt>.D }, <Pg>, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

### Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## STNT1H (scalar plus immediate, consecutive registers)

Contiguous store non-temporal of halfwords from multiple consecutive vectors (immediate index)

Contiguous store non-temporal of halfwords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	1	1	0	imm4				0	0	1	PNg				Rn				Zt				1
																	msz<1>msz<0>													N	

STNT1H { <Zt1>.H-<Zt2>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 16;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	1	0	0	0	0	0	0	1	1	0	imm4			1	0	1	PNg			Rn				Zt			0	1			
																	msz<1>msz<0>													N		

STNT1H { <Zt1>.H-<Zt4>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 16;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.
- For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1H (scalar plus immediate, single register)

Contiguous store non-temporal halfwords from vector (immediate index)

Contiguous store non-temporal of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	1	0	0	1	imm4				1	1	1	Pg			Rn				Zt					
msz<1>msz<0>																															

**STNT1H** { <Zt>.H }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g, PL];
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1H (scalar plus scalar, consecutive registers)

Contiguous store non-temporal of halfwords from multiple consecutive vectors (scalar index)

Contiguous store non-temporal of halfwords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm				0	0	1	PNg				Rn				Zt				1	
																msz<1>msz<0>														N	

STNT1H { <Zt1>.H-<Zt2>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 16;
```

### Four registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm				1	0	1	PNg				Rn				Zt				0	1
																msz<1>msz<0>														N	

STNT1H { <Zt1>.H-<Zt4>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 16;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.
- For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1H (scalar plus scalar, single register)

Contiguous store non-temporal halfwords from vector (scalar index)

Contiguous store non-temporal of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	1	0	0	Rm				0	1	1	Pg			Rn				Zt						
msz<1>msz<0>																															

**STNT1H** { <Zt>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(VL) src;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



STNT1H (vector plus scalar)

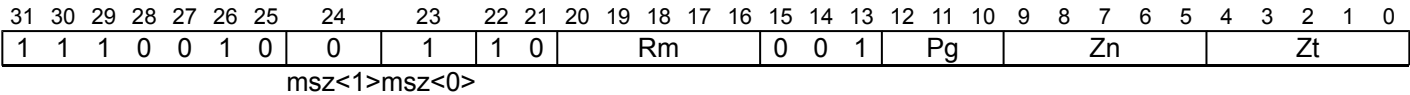
Scatter store non-temporal halfwords

Scatter store non-temporal of halfwords from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory. A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit unscaled offset](#) and [64-bit unscaled offset](#)

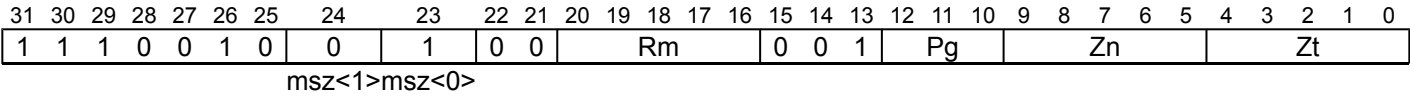
32-bit unscaled offset



STNT1H { <Zt>.S }, <Pg>, [<Zn>.S{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
```

64-bit unscaled offset



STNT1H { <Zt>.D }, <Pg>, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1W (scalar plus immediate, consecutive registers)

Contiguous store non-temporal of words from multiple consecutive vectors (immediate index)

Contiguous store non-temporal of words from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	1	1	0	imm4				0	1	0	PNg				Rn				Zt				1
																	msz<1>msz<0>													N	

```
STNT1W { <Zt1>.S-<Zt2>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 32;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	1	0	0	0	0	0	0	1	1	0	imm4				1	1	0	PNg				Rn				Zt				0	1
																	msz<1>msz<0>													N		

```
STNT1W { <Zt1>.S-<Zt4>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 32;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.
- For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1W (scalar plus immediate, single register)

Contiguous store non-temporal words from vector (immediate index)

Contiguous store non-temporal of words from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	1	0	0	0	1	imm4				1	1	1	Pg			Rn				Zt					
msz<1>msz<0>																															

**STNT1W** { <Zt>.S }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g, PL];
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1W (scalar plus scalar, consecutive registers)

Contiguous store non-temporal of words from multiple consecutive vectors (scalar index)

Contiguous store non-temporal of words from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

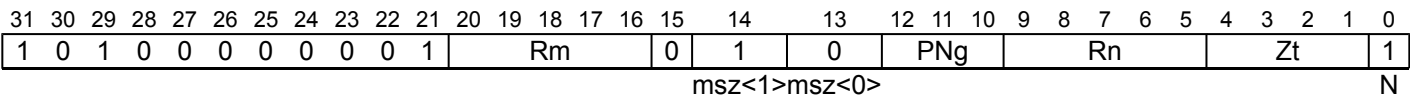
Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SVE2p1)

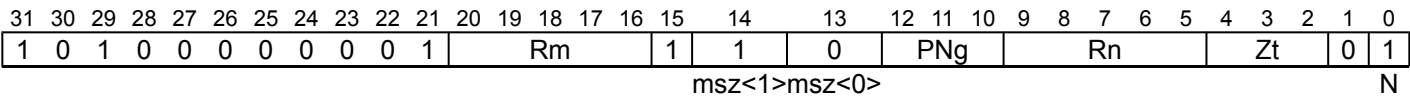


STNT1W { <Zt1>.S-<Zt2>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 32;
```

### Four registers

(FEAT\_SVE2p1)



STNT1W { <Zt1>.S-<Zt4>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 32;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.
- For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.
- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## STNT1W (scalar plus scalar, single register)

Contiguous store non-temporal words from vector (scalar index)

Contiguous store non-temporal of words from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	1	0	0	0	Rm				0	1	1	Pg			Rn				Zt						
msz<1>msz<0>																															

**STNT1W** { <Zt>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

```

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;

```

### Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

### Operation

```

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(VL) src;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];
        src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize];

```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STNT1W (vector plus scalar)

Scatter store non-temporal words

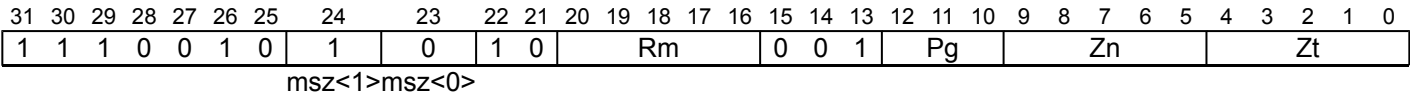
Scatter store non-temporal of words from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 2 classes: [32-bit unscaled offset](#) and [64-bit unscaled offset](#)

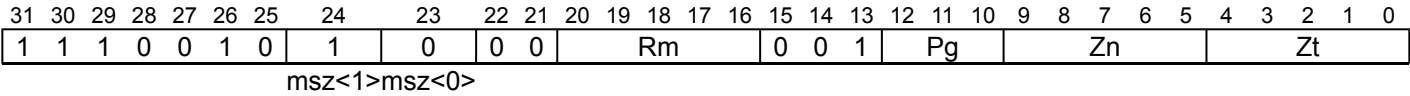
32-bit unscaled offset



STNT1W { <Zt>.S }, <Pg>, [<Zn>.S{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
```

64-bit unscaled offset



STNT1W { <Zt>.D }, <Pg>, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
```

Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
boolean contiguous = FALSE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize]<msize-1:0>;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# STR (predicate)

Store predicate register

Store a predicate register to a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current predicate register size in bytes. This instruction is unpredicated. The store is performed as contiguous byte accesses, each containing 8 consecutive predicate bits in ascending element order, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then a general-purpose base register must be aligned to 2 bytes. For programmer convenience, an assembler must also accept a predicate-as-counter register name for the source predicate register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	1	1	0	imm9h						0	0	0	imm9l			Rn			0	Pt					

STR <Pt>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Pt);
integer n = UInt(Rn);
integer imm = Sint(imm9h:imm9l);
```

## Assembler Symbols

- <Pt> Is the name of the scalable predicate transfer register, encoded in the "Pt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

## Operation

```
CheckSVEEnabled();
constant integer PL = CurrentVL DIV 8;
constant integer elements = PL DIV 8;
bits(PL) src;
bits(64) base;
integer offset = imm * elements;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if n == 31 then
    CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

src = P[t, PL];

boolean aligned = IsAligned(base + offset, 2);

if !aligned && AlignmentEnforced() then
    AArch64.Abort(base + offset, AlignmentFault(accdesc));

for e = 0 to elements-1
    AArch64.MemSingle(base + offset, 1, accdesc, aligned) = Elem[src, e, 8];
    offset = offset + 1;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



# STR (vector)

Store vector register

Store a vector register to a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current vector register size in bytes. This instruction is unpredicated. The store is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	1	1	0	imm9h						0	1	0	imm9l			Rn					Zt				

STR <Zt>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);
```

## Assembler Symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <imm> Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 8;
bits(VL) src;
bits(64) base;
integer offset = imm * elements;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if n == 31 then
    CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

src = Z[t, VL];

boolean aligned = IsAligned(base + offset, 16);

if !aligned && AlignmentEnforced() then
    AArch64.Abort(base + offset, AlignmentFault(accdesc));

for e = 0 to elements-1
    AArch64.MemSingle(base + offset, 1, accdesc, aligned) = Elem[src, e, 8];
    offset = offset + 1;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.





SUB (immediate)

Subtract immediate (unpredicated)

Subtract an unsigned immediate from each element of the source vector, and destructively place the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	0	0	0	0	1	1	1	sh	imm8								Zdn					

SUB <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    Elem[result, e, esize] = element1 - imm;

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SUB (vectors, predicated)

Subtract vectors (predicated)

Subtract active elements of the second source vector from corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	0	0	0	0	1	0	0	0	Pg	Zm			Zdn									

SUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

### Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```

### Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUB (vectors, unpredicated)

Subtract vectors (unpredicated)

Subtract all elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	0	0	0	1	0	0	size	1	Zm						0	0	0	0	0	1	Zn						Zd					

```
SUB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = element1 - element2;

Z[d, VL] = result;
```

Operational information

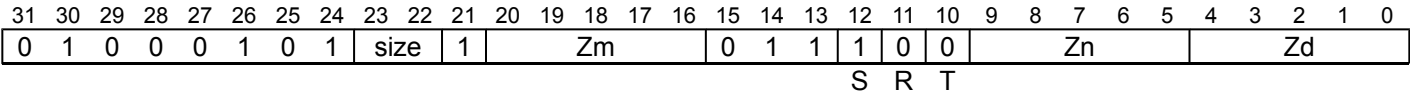
If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

SUBHNB

Subtract narrow high part (bottom)

Subtract each vector element of the second source vector from the corresponding vector element in the first source vector, and place the most significant half of the result in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. This instruction is unpredicated.



SUBHNB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = (element1 - element2) >> halfesize;
    Elem[result, 2*e + 0, halfesize] = res<halfesize-1:0>;
    Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

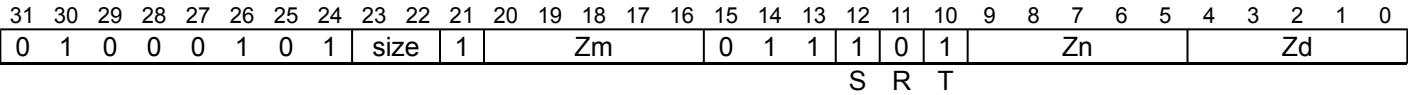
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUBHNT

Subtract narrow high part (top)

Subtract each vector element of the second source vector from the corresponding vector element in the first source vector, and place the most significant half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.



SUBHNT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = (element1 - element2) >> halfesize;
    Elem[result, 2*e + 1, halfesize] = res<halfesize-1:0>;

Z[d, VL] = result;
```



## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUBR (immediate)

Reversed subtract from immediate (unpredicated)

Reversed subtract from an unsigned immediate each element of the source vector, and destructively place the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	0	0	0	1	1	1	1	1	sh	imm8								Zdn				

SUBR <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    Elem[result, e, esize] = (imm - element1)<esize-1:0>;

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUBR (vectors)

Reversed subtract vectors (predicated)

Reversed subtract active elements of the first source vector from corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	0	0	0	1	1	0	0	0	Pg	Zm				Zdn								

```
SUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  if ActivePredicateElement(mask, e, esize) then
    Elem[result, e, esize] = element2 - element1;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

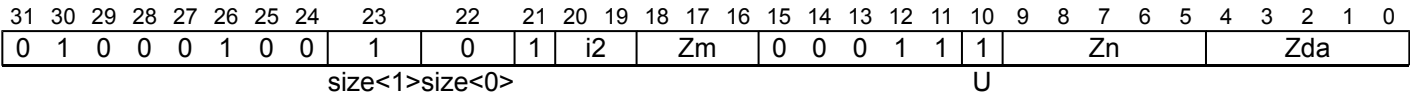
# SUDOT

Signed by unsigned integer indexed dot product

The signed by unsigned integer indexed dot product instruction computes the dot product of a group of four signed 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four unsigned 8-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated. ID\_AA64ZFR0\_EL1.I8MM indicates whether this instruction is implemented.

## SVE (FEAT\_I8MM)



SUDOT <Zda>.S, <Zn>.B, <Zm>.B[<imm>]

```
if (!HaveSVE() && !HaveSME()) || !HaveInt8MatMulExt() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

## Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

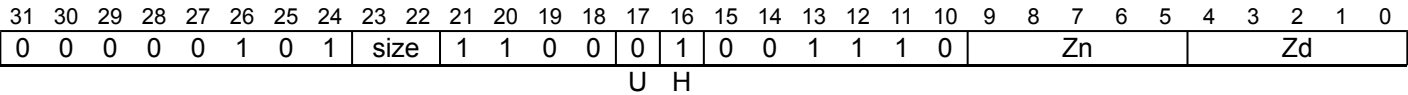
SUNPKHI, SUNPKLO

Signed unpack and extend half of vector

Unpack elements from the lowest or highest half of the source vector and then sign-extend them to place in elements of twice their size within the destination vector. This instruction is unpredicated.

It has encodings from 2 classes: [High half](#) and [Low half](#)

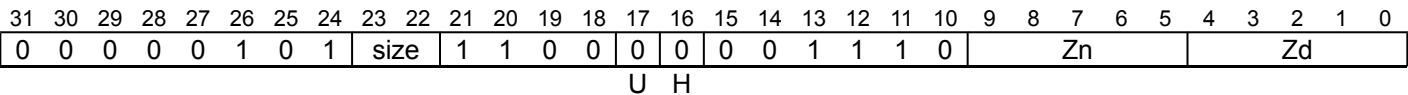
High half



SUNPKHI <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
boolean hi = TRUE;
```

Low half



SUNPKLO <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
boolean hi = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer hsize = esize DIV 2;
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(hsize) element = if hi then Elem[operand, e + elements, hsize] else Elem[operand, e, hsize];
    Elem[result, e, esize] = Extend(element, esize, unsigned);

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

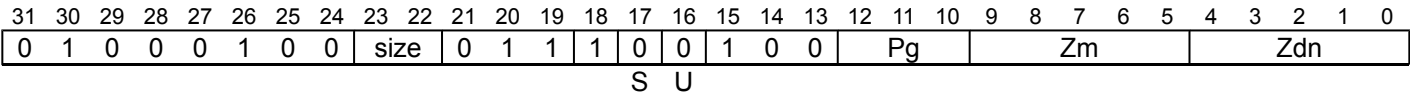
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUQADD

Signed saturating addition of unsigned value

Add active unsigned elements of the source vector to the corresponding signed elements of the addend vector, and destructively place the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . Inactive elements in the destination vector register remain unmodified.



```
SUQADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = SignedSat(SInt(element1) + UInt(element2), esize);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

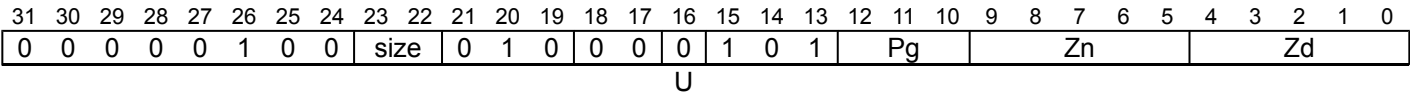
SXTB, SXTH, SXTW

Signed byte / halfword / word extend (predicated)

Sign-extend the least-significant sub-element of each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

It has encodings from 3 classes: [Byte](#) , [Halfword](#) and [Word](#)

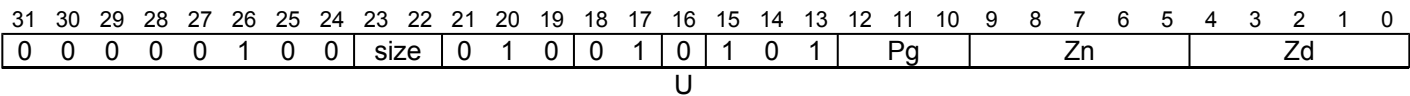
Byte



SXTB <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 8;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
```

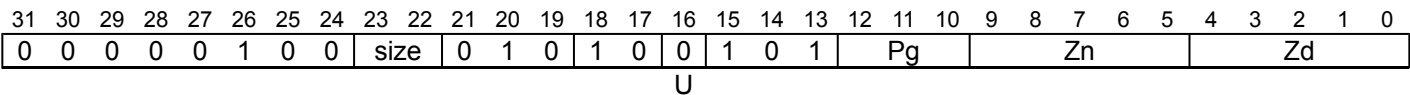
Halfword



SXTH <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
```

Word



SXTW <Zd>.D, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size != '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <T> For the byte variant: is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

For the halfword variant: is the size specifier, encoded in "size<0>":

size<0>	<T>
0	S
1	D

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = Extend(element<s_ esize-1:0>, esize, unsigned);

Z[d, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

TBL

Programmable table lookup in one or two vector table (zeroing)

Reads each element of the second source (index) vector and uses its value to select an indexed element from a table of elements consisting of one or two consecutive vector registers, where the first or only vector holds the lower numbered elements, and places the indexed table element in the destination vector element corresponding to the index vector element. If an index value is greater than or equal to the number of vector elements then it places zero in the corresponding destination vector element.

Since the index values can select any element in a vector this operation is not naturally vector length agnostic. It has encodings from 2 classes: [SVE](#) and [SVE2](#)

SVE

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	Zm						0	0	1	1	0	0	Zn						Zd			

TBL <Zd>.<T>, { <Zn>.<T> }, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean double_table = FALSE;
```

SVE2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	0	0	0	1	0	1	size	1	Zm						0	0	1	0	1	0	Zn						Zd					

TBL <Zd>.<T>, { <Zn1>.<T>, <Zn2>.<T> }, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean double_table = TRUE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded in the "Zn" field.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) indexes = Z[m, VL];
bits(VL) result;
constant integer table_size = if double_table then VL*2 else VL;
constant integer table_elems = table_size DIV esize;
bits(table_size) table;

if double_table then
    bits(VL) top = Z[(n + 1) MOD 32, VL];
    bits(VL) bottom = Z[n, VL];
    table = (top:bottom)<table_size-1:0>;
else
    table = Z[n, table_size];

for e = 0 to elements-1
    integer idx = UInt(Elem[indexes, e, esize]);
    Elem[result, e, esize] = if idx < table_elems then Elem[table, idx, esize] else Zeros(esize);

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TBLQ

Programmable table lookup within each quadword vector segment (zeroing)

For each 128-bit destination vector segment, reads each element of the corresponding second source (index) vector segment and uses its value to select an indexed element from the corresponding first source (table) vector segment. The indexed table element is placed in the element of the destination vector that corresponds to the index vector element. If an index value is greater than or equal to the number of elements in a 128-bit vector segment then it places zero in the corresponding destination vector element. This instruction is unpredicated.

SVE2  
(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		0	Zm				1	1	1	1	1	0	Zn				Zd						

TBLQ <Zd>.<T>, { <Zn>.<T> }, <Zm>.<T>

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elements = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for s = 0 to segments-1
  for e = 0 to elements-1
    integer idx = UInt(Elem[operand2, s * elements + e, esize]);
    if idx < elements then
      Elem[result, s * elements + e, esize] = Elem[operand1, s * elements + idx, esize];
    else
      Elem[result, s * elements + e, esize] = Zeros(esize);
Z[d, VL] = result;
```



## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TBX

Programmable table lookup in single vector table (merging)

Reads each element of the second source (index) vector and uses its value to select an indexed element from a table of elements in the first source vector, and places the indexed element in the destination vector element corresponding to the index vector element. If an index value is greater than or equal to the number of vector elements then the corresponding destination vector element is left unchanged.

Since the index values can select any element in a vector this operation is not naturally vector length agnostic.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	1	0	1	size	1	Zm						0	0	1	0	1	1	Zn						Zd				

TBX <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    integer element2 = UInt(Elem[operand2, e, esize]);
    if element2 < elements then
        Elem[result, e, esize] = Elem[operand1, element2, esize];

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



TBXQ

Programmable table lookup within each quadword vector segment (merging)

For each 128-bit destination vector segment, reads each element of the corresponding second source (index) vector segment and uses its value to select an indexed element from the corresponding first source (table) vector segment. The indexed table element is placed in the element of the destination vector that corresponds to the index vector element. If an index value is greater than or equal to the number of elements in a 128-bit vector segment then the corresponding destination vector element is left unchanged. This instruction is unpredicated.

SVE2  
(FEAT\_SVE2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1				Zm			0	0	1	1	0	1				Zn				Zd		

TBXQ <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elements = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];

for s = 0 to segments-1
  for e = 0 to elements-1
    integer idx = UInt(Elem[operand2, s * elements + e, esize]);
    if idx < elements then
      Elem[result, s * elements + e, esize] = Elem[operand1, s * elements + idx, esize];

Z[d, VL] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

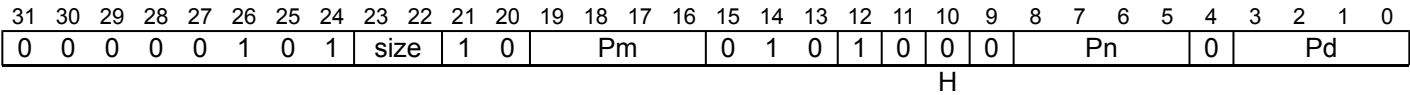
## TRN1, TRN2 (predicates)

Interleave even or odd elements from two predicates

Interleave alternating even or odd-numbered elements from the first and second source predicates and place in elements of the destination predicate. This instruction is unpredicated.

It has encodings from 2 classes: [Even](#) and [Odd](#)

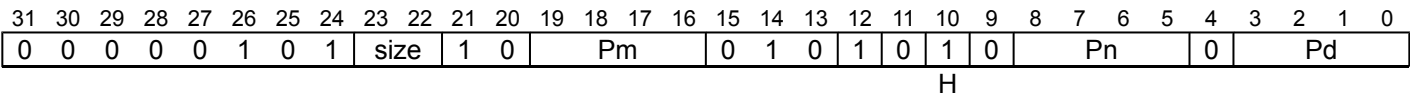
### Even



TRN1 [<Pd>.<T>](#), [<Pn>.<T>](#), [<Pm>.<T>](#)

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 0;
```

### Odd



TRN2 [<Pd>.<T>](#), [<Pn>.<T>](#), [<Pm>.<T>](#)

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 1;
```

### Assembler Symbols

- <Pd>

Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D
- <Pn>

Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm>

Is the name of the second source scalable predicate register, encoded in the "Pm" field.

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer pairs = VL DIV (esize * 2);  
bits(PL) operand1 = P[n, PL];  
bits(PL) operand2 = P[m, PL];  
bits(PL) result;  
  
for p = 0 to pairs-1  
    Elem[result, 2*p+0, esize DIV 8] = Elem[operand1, 2*p+part, esize DIV 8];  
    Elem[result, 2*p+1, esize DIV 8] = Elem[operand2, 2*p+part, esize DIV 8];  
  
P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## TRN1, TRN2 (vectors)

Interleave even or odd elements from two vectors

Interleave alternating even or odd-numbered elements from the first and second source vectors and place in elements of the destination vector. This instruction is unpredicated.

The 128-bit element variant requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero. ID\_AA64ZFR0\_EL1.F64MM indicates whether the 128-bit element variant is implemented. The 128-bit element variant is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 4 classes: [Even](#) , [Even \(quadwords\)](#) , [Odd](#) and [Odd \(quadwords\)](#)

### Even

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	Zm						0	1	1	1	0	0	Zn						Zd			
																										H					

TRN1 [<Zd>.<T>](#), [<Zn>.<T>](#), [<Zm>.<T>](#)

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

### Even (quadwords)

(FEAT\_F64MM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	1	0	1	1	0	1	Zm						0	0	0	1	1	0	Zn						Zd			
																										H						

TRN1 [<Zd>.Q](#), [<Zn>.Q](#), [<Zm>.Q](#)

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

### Odd

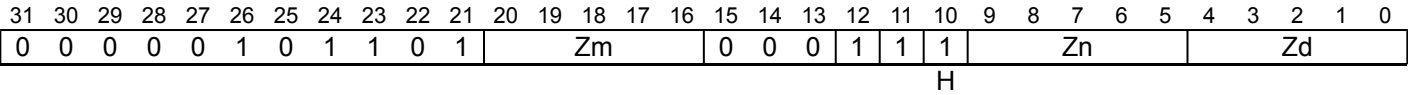
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	Zm						0	1	1	1	0	1	Zn						Zd			
																										H					

TRN2 [<Zd>.<T>](#), [<Zn>.<T>](#), [<Zm>.<T>](#)

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```



Odd (quadwords)  
(FEAT\_F64MM)



TRN2 <Zd>.Q, <Zn>.Q, <Zm>.Q

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
if VL < esize * 2 then UNDEFINED;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Zeros(VL);

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

Z[d, VL] = result;
```

Operational information

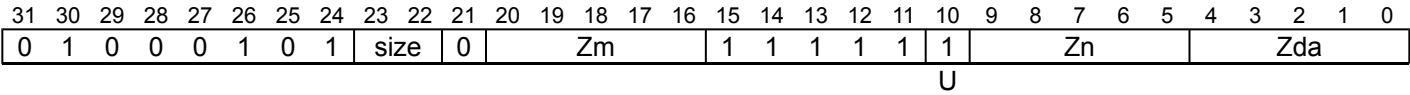
If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

UABA

Unsigned absolute difference and accumulate

Compute the absolute difference between unsigned integer values in elements of the second source vector and corresponding elements of the first source vector, and add the difference to the corresponding elements of the destination vector. This instruction is unpredicated.



UABA <Zda>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;

Z[da, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

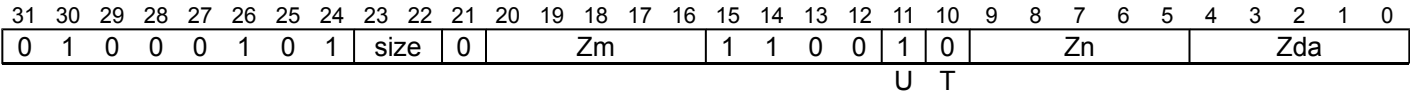
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UABALB

Unsigned absolute difference and accumulate long (bottom)

Compute the absolute difference between even-numbered unsigned elements of the second source vector and corresponding elements of the first source vector, and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.



UABALB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;

Z[da, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

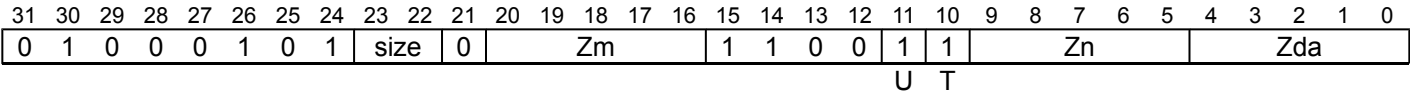
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UABALT

Unsigned absolute difference and accumulate long (top)

Compute the absolute difference between odd-numbered unsigned elements of the second source vector and corresponding elements of the first source vector, and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.



```
UABALT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;

Z[da, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

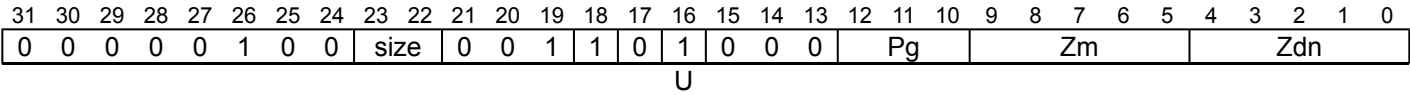
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UABD

Unsigned absolute difference (predicated)

Compute the absolute difference between unsigned integer values in active elements of the second source vector and corresponding elements of the first source vector and destructively place the difference in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



```
UABD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if ActivePredicateElement(mask, e, esize) then
        integer absdiff = Abs(element1 - element2);
        Elem[result, e, esize] = absdiff<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:



- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

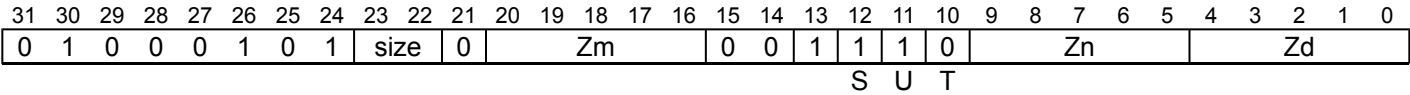
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UABDLB

Unsigned absolute difference long (bottom)

Compute the absolute difference between the even-numbered unsigned integer values in elements of the second source vector and the corresponding elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



```
UABDLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    integer res = Abs(element1 - element2);
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

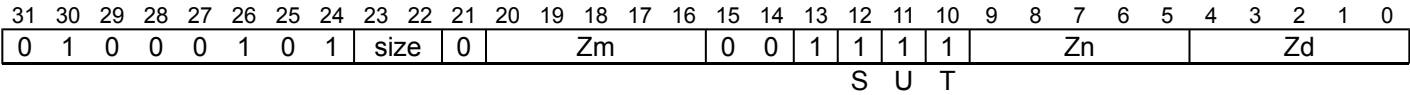
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UABDLT

Unsigned absolute difference long (top)

Compute the absolute difference between the odd-numbered unsigned integer values in elements of the second source vector and corresponding elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



```
UABDLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    integer res = Abs(element1 - element2);
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

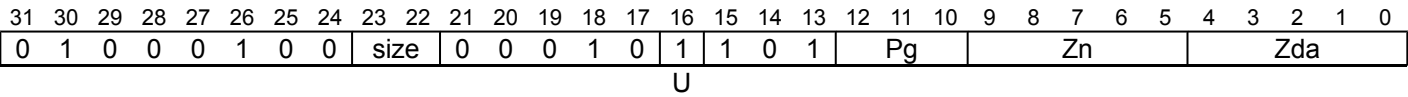
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UADALP

Unsigned add and accumulate long pairwise

Add pairs of adjacent unsigned integer values and accumulate the results into the overlapping double-width elements of the destination vector.



UADALP <Zda>.<T>, <Pg>/M, <Zn>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the second source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand_acc = Z[da, VL];
bits(VL) operand_src = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if !ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Elem[operand_acc, e, esize];
    else
        integer element1 = UInt(Elem[operand_src, 2*e + 0, esize DIV 2]);
        integer element2 = UInt(Elem[operand_src, 2*e + 1, esize DIV 2]);
        bits(esize) sum = (element1 + element2)<esize-1:0>;
        Elem[result, e, esize] = Elem[operand_acc, e, esize] + sum;

Z[da, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

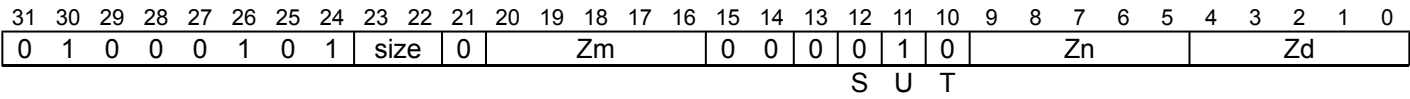
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UADDLB

Unsigned add long (bottom)

Add the corresponding even-numbered unsigned elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



UADDLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 0;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 + element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```



## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

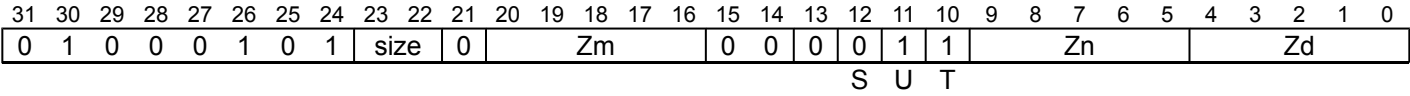
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UADDLT

Unsigned add long (top)

Add the corresponding odd-numbered unsigned elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



UADDLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 1;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 + element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

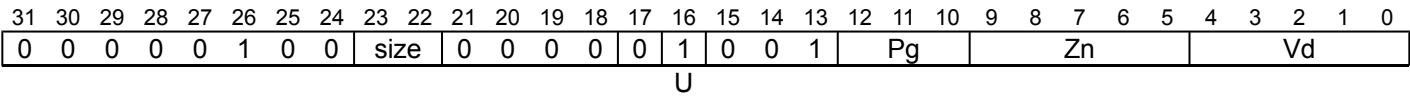
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UADDV

Unsigned add reduction to scalar

Unsigned add horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Narrow elements are first zero-extended to 64 bits. Inactive elements in the source vector are treated as zero.



```
UADDV <Dd>, <Pg>, <Zn>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

- <Dd> Is the 64-bit name of the destination SIMD&FP register, encoded in the "Vd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
integer sum = 0;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer element = UInt(Elem[operand, e, esize]);
        sum = sum + element;

V[d, 64] = sum<63:0>;
```

Operational information

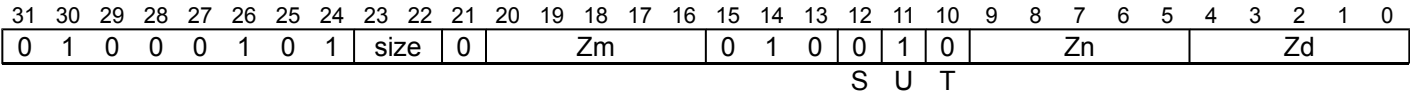
- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.



UADDWB

Unsigned add wide (bottom)

Add the even-numbered unsigned elements of the second source vector to the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.



```
UADDWB <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    Elem[result, e, esize] = (element1 + element2)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

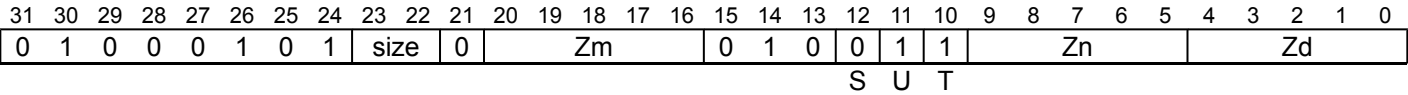
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UADDWT

Unsigned add wide (top)

Add the odd-numbered unsigned elements of the second source vector to the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.



```
UADDWT <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    Elem[result, e, esize] = (element1 + element2)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:



- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

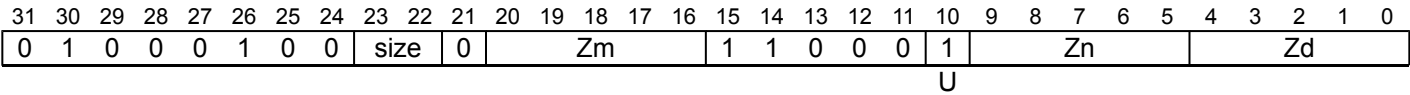
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UCLAMP

Unsigned clamp to minimum/maximum vector

Clamp each unsigned element in the destination vector to between the unsigned minimum value in the corresponding element of the first source vector and the unsigned maximum value in the corresponding element of the second source vector and destructively write the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE2  
(FEAT\_SVE2p1)



```
UCLAMP <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSME() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[d, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer element3 = UInt(Elem[operand3, e, esize]);
    integer res = Min(Max(element1, element3), element2);
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UCVTF

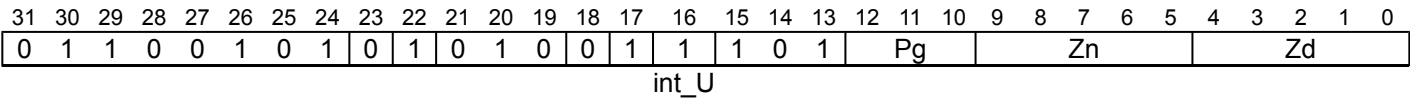
Unsigned integer convert to floating-point (predicated)

Convert to floating-point from the unsigned integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.

It has encodings from 7 classes: [16-bit to half-precision](#) , [32-bit to half-precision](#) , [32-bit to single-precision](#) , [32-bit to double-precision](#) , [64-bit to half-precision](#) , [64-bit to single-precision](#) and [64-bit to double-precision](#)

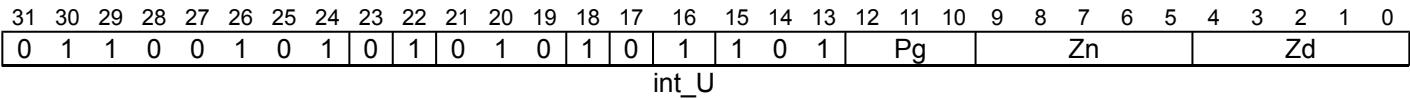
16-bit to half-precision



UCVTF <Zd>.H, <Pg>/M, <Zn>.H

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 16;
constant integer d_esign = 16;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

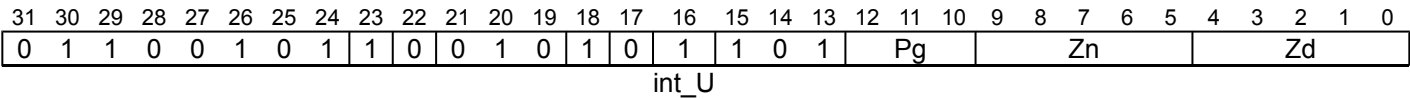
32-bit to half-precision



UCVTF <Zd>.H, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 32;
constant integer d_esign = 16;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

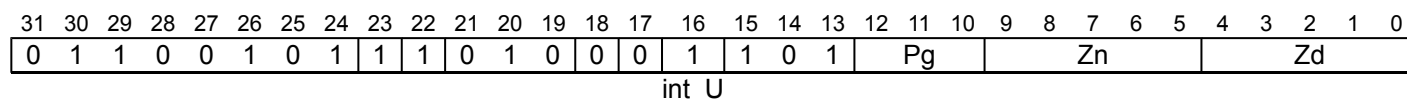
32-bit to single-precision



UCVTF <Zd>.S, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 32;
constant integer d_esign = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

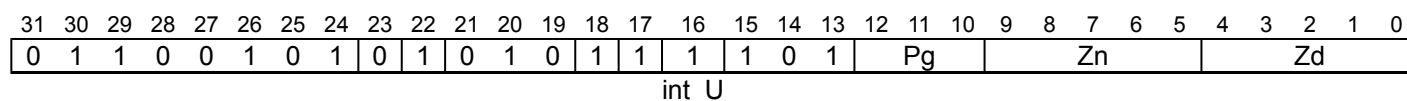
### 32-bit to double-precision



UCVTF <Zd>.D, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 32;
constant integer d_esign = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

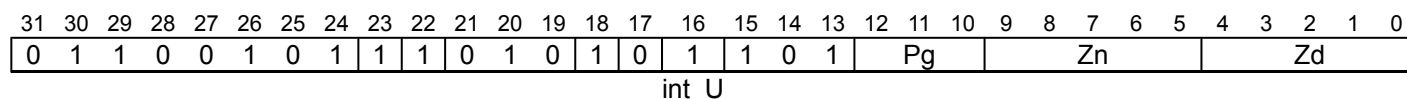
### 64-bit to half-precision



UCVTF <Zd>.H, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esign = 64;
constant integer d_esign = 16;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

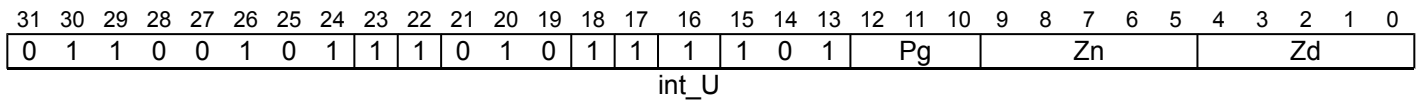
### 64-bit to single-precision



UCVTF <Zd>.S, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

## 64-bit to double-precision



UCVTF <Zd>.D, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

## Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        bits(d_esize) fpval = FixedToFP(element<s_esize-1:0>, 0, unsigned, FPCR[], rounding, d_esize);
        Elem[result, e, esize] = ZeroExtend(fpval, esize);

Z[d, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

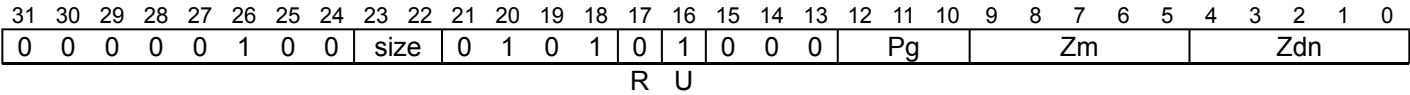
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UDIV

Unsigned divide (predicated)

Unsigned divide active elements of the first source vector by corresponding elements of the second source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



```
UDIV <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size<0>":

size<0>	<T>
0	S
1	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if ActivePredicateElement(mask, e, esize) then
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer quotient;
        if element2 == 0 then
            quotient = 0;
        else
            quotient = RoundTowardsZero(Real(element1) / Real(element2));
        Elem[result, e, esize] = quotient<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```



## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

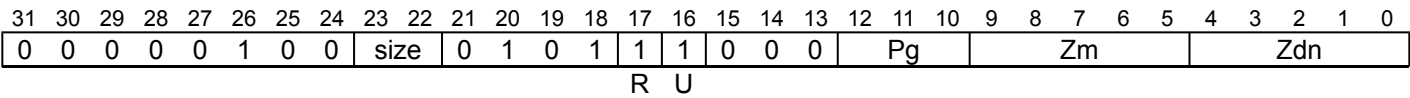
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UDIVR

Unsigned reversed divide (predicated)

Unsigned reversed divide active elements of the second source vector by corresponding elements of the first source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



```
UDIVR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size<0>":

size<0>	<T>
0	S
1	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if ActivePredicateElement(mask, e, esize) then
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer quotient;
        if element1 == 0 then
            quotient = 0;
        else
            quotient = RoundTowardsZero(Real(element2) / Real(element1));
        Elem[result, e, esize] = quotient<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

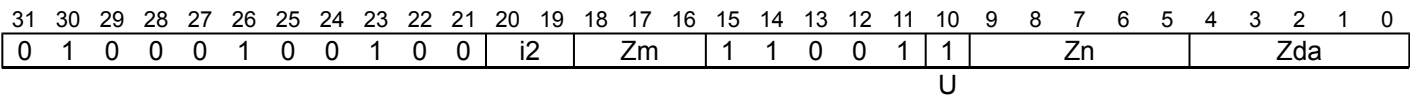
## UDOT (2-way, indexed)

Unsigned integer indexed dot product

The unsigned integer indexed dot product instruction computes the dot product of a group of two unsigned 16-bit integer values held in each 32-bit element of the first source vector multiplied by a group of two unsigned 16-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated.

### SVE2 (FEAT\_SVE2p1)



UDOT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index of a group of two 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 1
        integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
        integer element2 = UInt(Elem[operand2, 2 * s + i, esize DIV 2]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

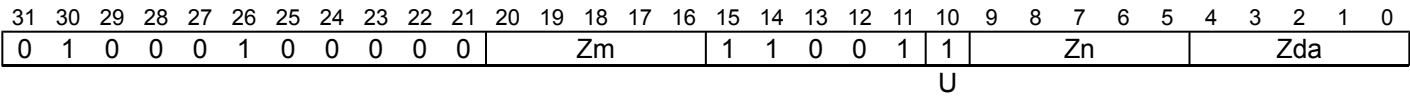
## UDOT (2-way, vectors)

Unsigned integer dot product

The unsigned integer dot product instruction computes the dot product of a group of two unsigned 16-bit integer values held in each 32-bit element of the first source vector multiplied by a group of two unsigned 16-bit integer values in the corresponding 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

This instruction is unpredicated.

### SVE2 (FEAT\_SVE2p1)



UDOT <Zda>.S, <Zn>.H, <Zm>.H

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
  bits(esize) res = Elem[operand3, e, esize];
  for i = 0 to 1
    integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2 * e + i, esize DIV 2]);
    res = res + element1 * element2;
  Elem[result, e, esize] = res;

Z[da, VL] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

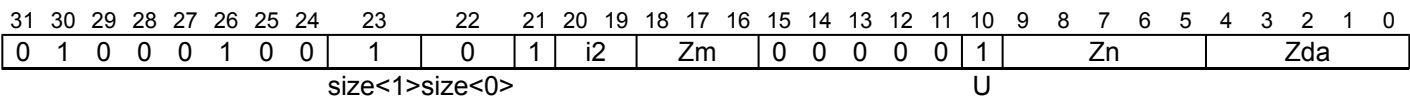


## UDOT (4-way, indexed)

Unsigned integer indexed dot product

The unsigned integer indexed dot product instruction computes the dot product of a group of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four unsigned 8-bit or 16-bit integer values in an indexed 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector. The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group. This instruction is unpredicated. It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

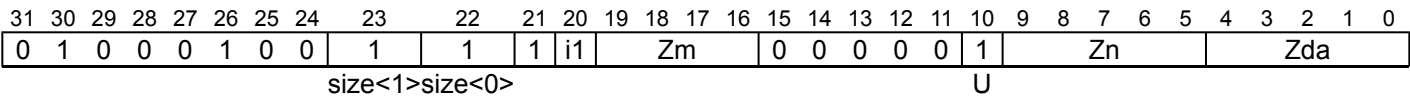
### 32-bit



UDOT <Zda>.S, <Zn>.B, <Zm>.B[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### 64-bit



UDOT <Zda>.D, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.  
For the 64-bit variant: is the immediate index of a 64-bit group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the "i1" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

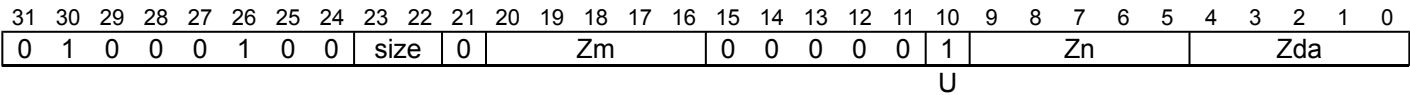
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UDOT (4-way, vectors)

Unsigned integer dot product

The unsigned integer dot product instruction computes the dot product of a group of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four unsigned 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector. This instruction is unpredicated.



UDOT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size<0>":

size<0>	<T>
0	S
1	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size<0>":

size<0>	<Tb>
0	B
1	H

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
  bits(esize) res = Elem[operand3, e, esize];
  for i = 0 to 3
    integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
    integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
    res = res + element1 * element2;
  Elem[result, e, esize] = res;

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

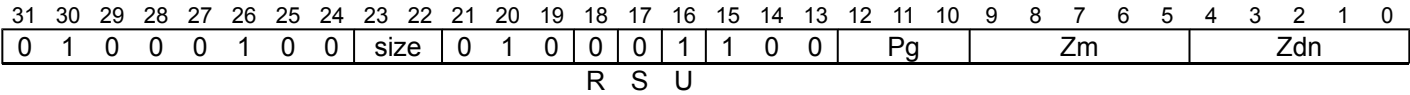
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# UHADD

Unsigned halving addition

Add active unsigned elements of the first source vector to corresponding unsigned elements of the second source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



```
UHADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

## Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = (element1 + element2) >> 1;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```

## Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

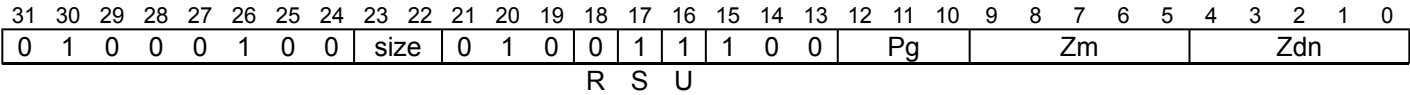
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UHSUB

Unsigned halving subtract

Subtract active unsigned elements of the second source vector from corresponding unsigned elements of the first source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



```
UHSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = (element1 - element2) >> 1;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

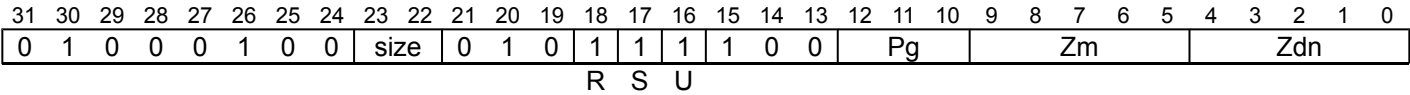
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UHSUBR

Unsigned halving subtract reversed vectors

Subtract active unsigned elements of the first source vector from corresponding unsigned elements of the second source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



```
UHSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = (element2 - element1) >> 1;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.



- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

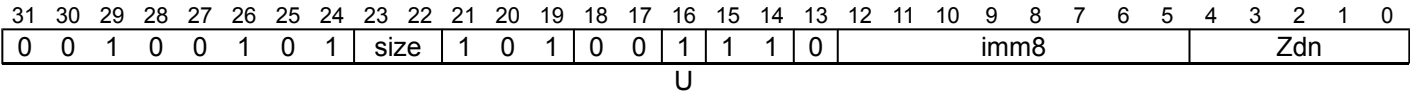
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMAX (immediate)

Unsigned maximum with immediate (unpredicated)

Determine the unsigned maximum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is an unsigned 8-bit value in the range 0 to 255, inclusive. This instruction is unpredicated.



```
UMAX <Zdn>.<T>, <Zdn>.<T>, #<imm>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;
integer imm = Int(imm8, unsigned);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm> Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    Elem[result, e, esize] = Max(element1, imm)<esize-1:0>;

Z[dn, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

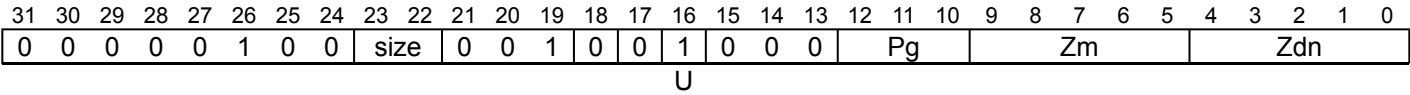
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# UMAX (vectors)

Unsigned maximum vectors (predicated)

Determine the unsigned maximum of active elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



```
UMAX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

## Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if ActivePredicateElement(mask, e, esize) then
        integer maximum = Max(element1, element2);
        Elem[result, e, esize] = maximum<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

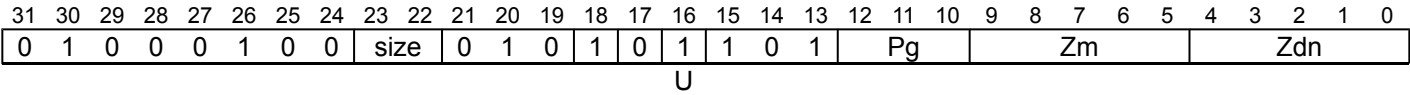
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMAXP

Unsigned maximum pairwise

Compute the maximum value of each pair of adjacent unsigned integer elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.



```
UMAXP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
integer element1;
integer element2;

for e = 0 to elements-1
  if !ActivePredicateElement(mask, e, esize) then
    Elem[result, e, esize] = Elem[operand1, e, esize];
  else
    if IsEven(e) then
      element1 = UInt(Elem[operand1, e + 0, esize]);
      element2 = UInt(Elem[operand1, e + 1, esize]);
    else
      element1 = UInt(Elem[operand2, e - 1, esize]);
      element2 = UInt(Elem[operand2, e + 0, esize]);
    integer res = Max(element1, element2);
    Elem[result, e, esize] = res<esize-1:0>;

Z[dn, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

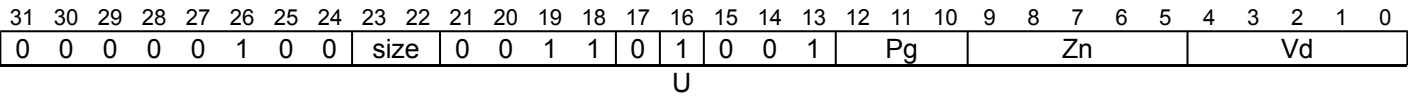
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# UMAXQV

Unsigned maximum reduction of quadword vector segments

Unsigned maximum of the same element numbers from each 128-bit source vector segment, placing each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as zero.

## SVE2 (FEAT\_SVE2p1)



UMAXQV <Vd>.<T>, <Pg>, <Zn>.<Tb>

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = TRUE;
```

### Assembler Symbols

<Vd> Is the name of the destination SIMD&FP register, encoded in the "Vd" field.

<T> Is an arrangement specifier, encoded in "size":

size	<T>
00	16B
01	8H
10	4S
11	2D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	B
01	H
10	S
11	D



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elemperssegment = 128 DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(128) result = Zeros(128);
bits(128) stmp = Zeros(128);

integer dtmp;

for e = 0 to elemperssegment-1
    dtmp = if unsigned then 0 else -(2^(esize-1));
    for s = 0 to segments-1
        if ActivePredicateElement(mask, s * elemperssegment + e, esize) then
            stmp = Elem[operand, s, 128];
            dtmp = Max(dtmp, UInt(Elem[stmp, e, esize]));
        Elem[result, e, esize] = dtmp<esize-1:0>;

V[d, 128] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

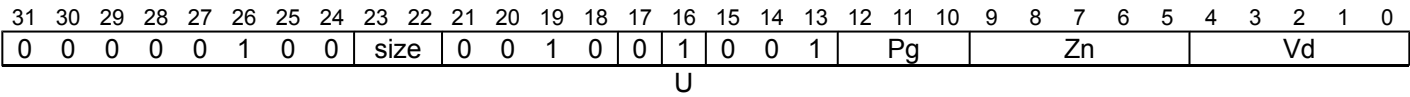
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMAXV

Unsigned maximum reduction to scalar

Unsigned maximum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as zero.



UMAXV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = TRUE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
integer maximum = if unsigned then 0 else -(2^(esize-1));

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer element = Int(Elem[operand, e, esize], unsigned);
        maximum = Max(maximum, element);

V[d, esize] = maximum<esize-1:0>;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

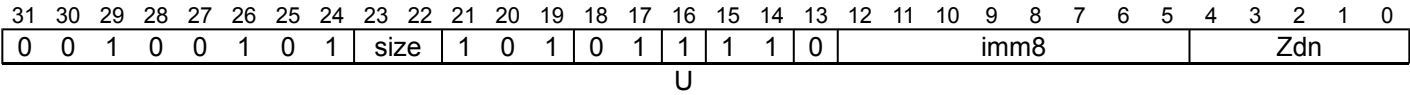
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMIN (immediate)

Unsigned minimum with immediate (unpredicated)

Determine the unsigned minimum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is an unsigned 8-bit value in the range 0 to 255, inclusive. This instruction is unpredicated.



```
UMIN <Zdn>.<T>, <Zdn>.<T>, #<imm>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;
integer imm = Int(imm8, unsigned);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm> Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    Elem[result, e, esize] = Min(element1, imm)<esize-1:0>;

Z[dn, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

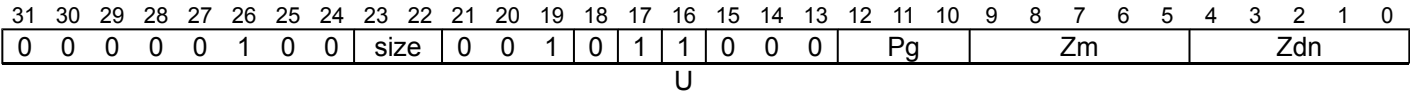
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMIN (vectors)

Unsigned minimum vectors (predicated)

Determine the unsigned minimum of active elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



```
UMIN <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if ActivePredicateElement(mask, e, esize) then
        integer minimum = Min(element1, element2);
        Elem[result, e, esize] = minimum<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

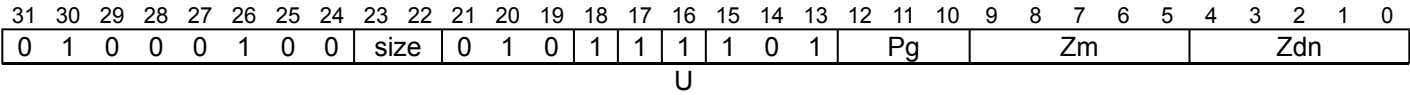
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMINP

Unsigned minimum pairwise

Compute the minimum value of each pair of adjacent unsigned integer elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.



```
UMINP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
integer element1;
integer element2;

for e = 0 to elements-1
  if !ActivePredicateElement(mask, e, esize) then
    Elem[result, e, esize] = Elem[operand1, e, esize];
  else
    if IsEven(e) then
      element1 = UInt(Elem[operand1, e + 0, esize]);
      element2 = UInt(Elem[operand1, e + 1, esize]);
    else
      element1 = UInt(Elem[operand2, e - 1, esize]);
      element2 = UInt(Elem[operand2, e + 0, esize]);
    integer res = Min(element1, element2);
    Elem[result, e, esize] = res<esize-1:0>;

Z[dn, VL] = result;
```



## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

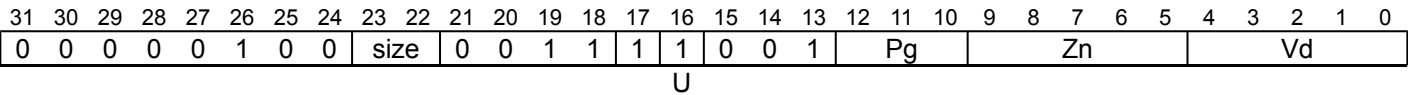
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMINQV

Unsigned minimum reduction of quadword vector segments

Unsigned minimum of the same element numbers from each 128-bit source vector segment, placing each result into the corresponding element number of the 128-bit SIMD&FP destination register. Inactive elements in the source vector are treated as the maximum unsigned integer for the element size.

SVE2  
(FEAT\_SVE2p1)



UMINQV <Vd>.<T>, <Pg>, <Zn>.<Tb>

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = TRUE;
```

Assembler Symbols

<Vd> Is the name of the destination SIMD&FP register, encoded in the "Vd" field.

<T> Is an arrangement specifier, encoded in "size":

size	<T>
00	16B
01	8H
10	4S
11	2D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	B
01	H
10	S
11	D

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elempersegment = 128 DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(128) result = Zeros(128);
bits(128) stmp = Zeros(128);

integer dtmp;

for e = 0 to elempersegment-1
    dtmp = if unsigned then (2^esize - 1) else (2^(esize-1) - 1);
    for s = 0 to segments-1
        if ActivePredicateElement(mask, s * elempersegment + e, esize) then
            stmp = Elem[operand, s, 128];
            dtmp = Min(dtmp, UInt(Elem[stmp, e, esize]));
        Elem[result, e, esize] = dtmp<esize-1:0>;

V[d, 128] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

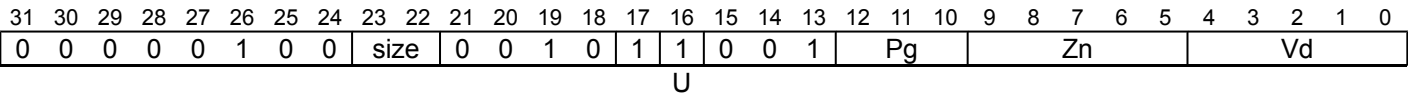
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMINV

Unsigned minimum reduction to scalar

Unsigned minimum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the maximum unsigned integer for the element size.



```
UMINV <V><d>, <Pg>, <Zn>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = TRUE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
integer minimum = if unsigned then (2^esize - 1) else (2^(esize-1) - 1);

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer element = Int(Elem[operand, e, esize], unsigned);
        minimum = Min(minimum, element);

V[d, esize] = minimum<esize-1:0>;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UMLALB (indexed)

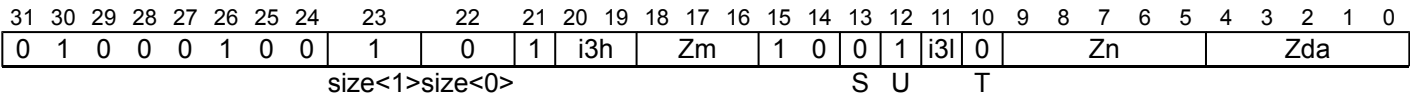
Unsigned multiply-add long to accumulator (bottom, indexed)

Multiply the even-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment and destructively add to the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

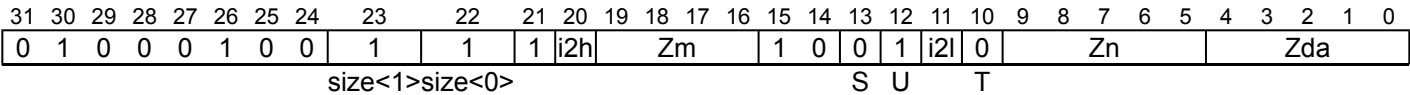
### 32-bit



UMLALB <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

### 64-bit



UMLALB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = UInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + product;

Z[da, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is

UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

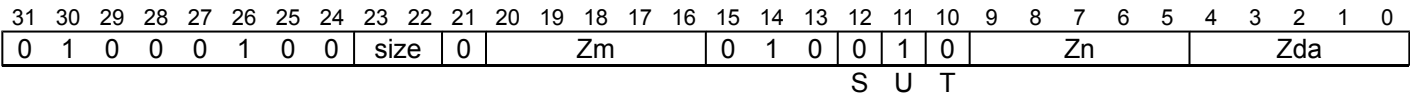
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMLALB (vectors)

Unsigned multiply-add long to accumulator (bottom)

Multiply the corresponding even-numbered unsigned elements of the first and second source vectors and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.



```
UMLALB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + product;

Z[da, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:



- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UMLALT (indexed)

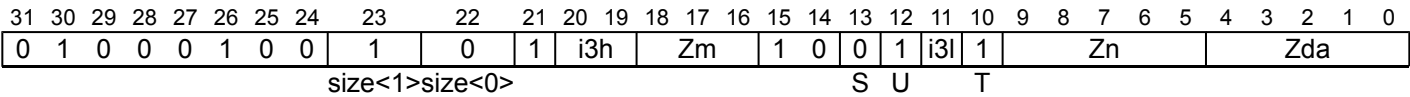
Unsigned multiply-add long to accumulator (top, indexed)

Multiply the odd-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment and destructively add to the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

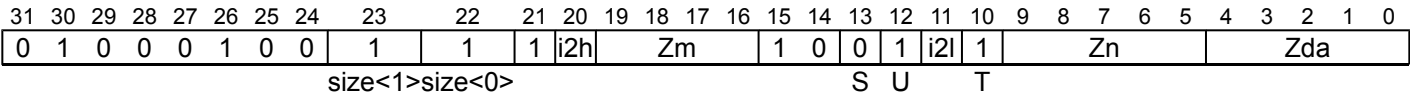
### 32-bit



**UMLALT** <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

### 64-bit



**UMLALT** <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = UInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2) < 2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + product;

Z[da, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

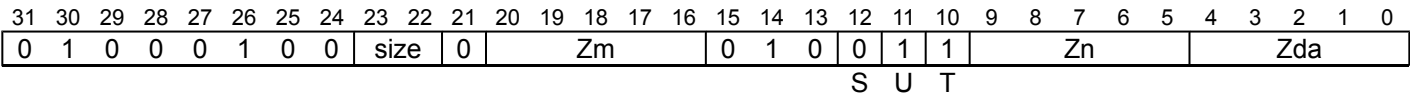
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMLALT (vectors)

Unsigned multiply-add long to accumulator (top)

Multiply the corresponding odd-numbered unsigned elements of the first and second source vectors and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.



```
UMLALT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + product;

Z[da, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMLSLB (indexed)

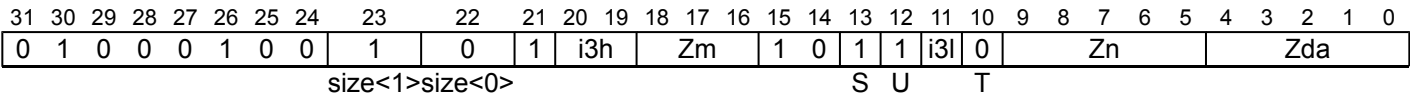
Unsigned multiply-subtract long from accumulator (bottom, indexed)

Multiply the even-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment and destructively subtract from the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: 32-bit and 64-bit

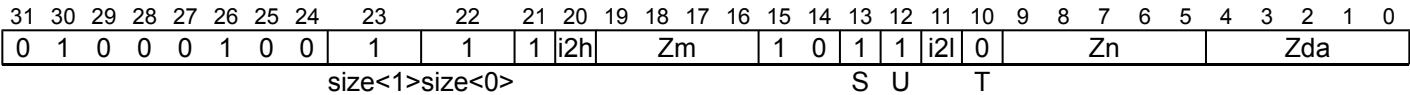
32-bit



UMLSLB <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

64-bit



UMLSLB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = UInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] - product;

Z[da, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

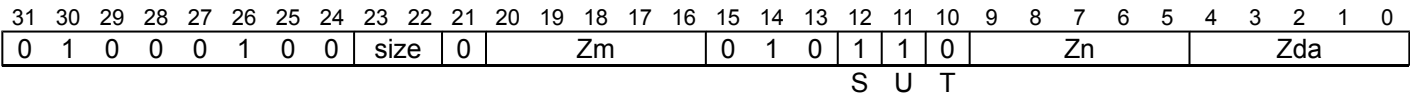
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMLSLB (vectors)

Unsigned multiply-subtract long from accumulator (bottom)

Multiply the corresponding even-numbered unsigned elements of the first and second source vectors and destructively subtract from the overlapping double-width elements of the addend vector. This instruction is unpredicated.



```
UMLSLB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] - product;

Z[da, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:



- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UMLSLT (indexed)

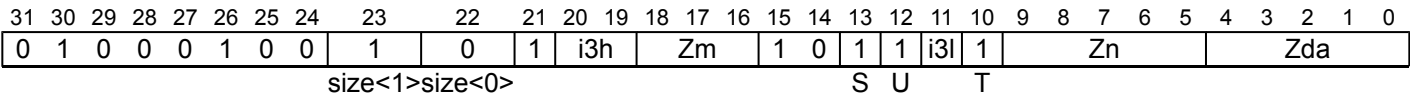
Unsigned multiply-subtract long from accumulator (top, indexed)

Multiply the odd-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment and destructively subtract from the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

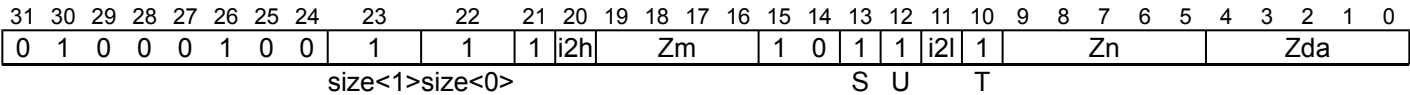
### 32-bit



UMLSLT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

### 64-bit



UMLSLT <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = UInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] - product;

Z[da, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

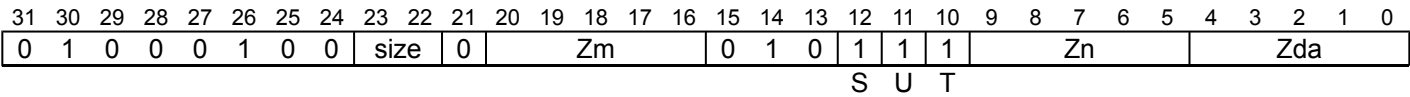
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMLSLT (vectors)

Unsigned multiply-subtract long from accumulator (top)

Multiply the corresponding odd-numbered unsigned elements of the first and second source vectors and destructively subtract from the overlapping double-width elements of the addend vector. This instruction is unpredicated.



UMLSLT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] - product;

Z[da, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMMLA

Unsigned integer matrix multiply-accumulate

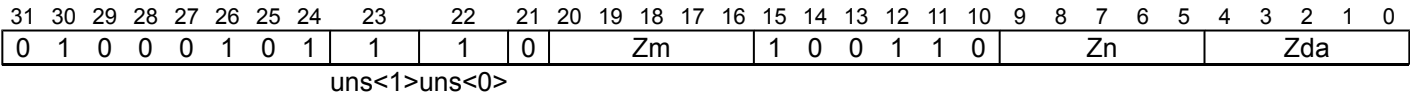
The unsigned integer matrix multiply-accumulate instruction multiplies the 2×8 matrix of unsigned 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of unsigned 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.I8MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE  
(FEAT\_I8MM)



```
UMMLA <Zda>.S, <Zn>.B, <Zm>.B
```

```
if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result = Zeros(VL);
bits(128) op1, op2;
bits(128) res, addend;

for s = 0 to segments-1
    op1 = Elem[operand1, s, 128];
    op2 = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];
    res = MatMulAdd(addend, op2, op1_unsigned, op2_unsigned);
    Elem[result, s, 128] = res;

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMULH (predicated)

Unsigned multiply returning high half (predicated)

Widening multiply unsigned integer values in active elements of the first source vector by corresponding elements of the second source vector and destructively place the high half of the result in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	0	0	1	1	0	0	0	Pg			Zm					Zdn					
																H		U													

```
UMULH <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if ActivePredicateElement(mask, e, esize) then
        integer product = (element1 * element2) >> esize;
        Elem[result, e, esize] = product<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:



- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is

UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

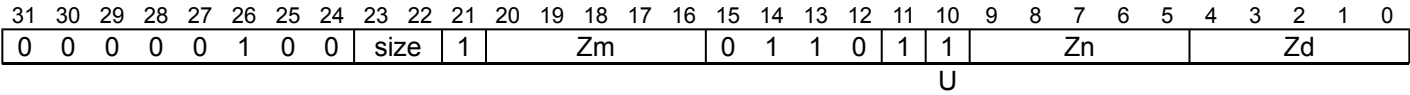
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMULH (unpredicated)

Unsigned multiply returning high half (unpredicated)

Widening multiply unsigned integer values of all elements of the first source vector by corresponding elements of the second source vector and place the high half of the result in the corresponding elements of the destination vector. This instruction is unpredicated.



```
UMULH <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    integer product = (element1 * element2) >> esize;
    Elem[result, e, esize] = product<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



## UMULLB (indexed)

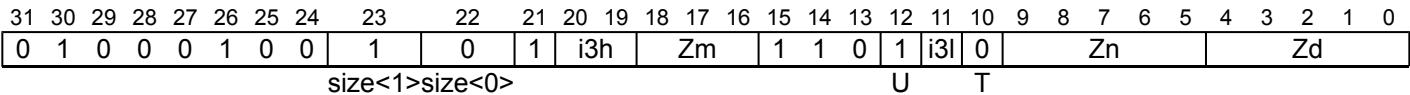
Unsigned multiply long (bottom, indexed)

Multiply the even-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment, and place the results in the overlapping double-width elements of the destination vector register.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

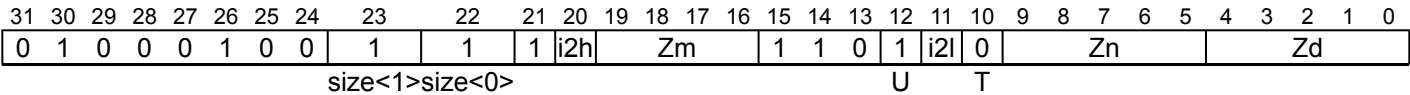
### 32-bit



UMULLB <Zd>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;
```

### 64-bit



UMULLB <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;
```

### Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = UInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
    integer res = element1 * element2;
    Elem[result, e, 2*esize] = res<2*esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

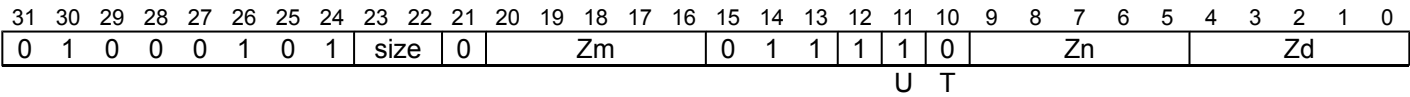
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMULLB (vectors)

Unsigned multiply long (bottom)

Multiply the corresponding even-numbered unsigned elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



```
UMULLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    integer res = element1 * element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMULLT (indexed)

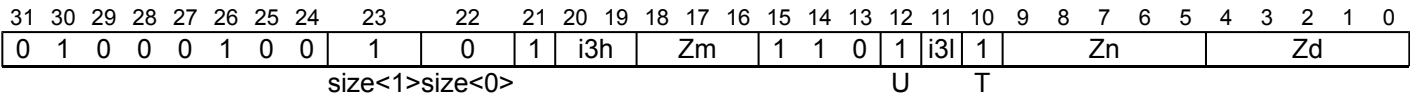
Unsigned multiply long (top, indexed)

Multiply the odd-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment, and place the results in the overlapping double-width elements of the destination vector register.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: 32-bit and 64-bit

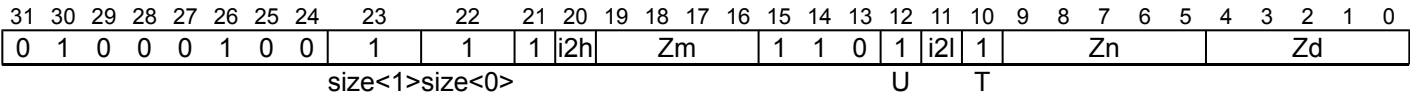
32-bit



```
UMULLT <Zd>.S, <Zn>.H, <Zm>.H[<imm>]
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;
```

64-bit



```
UMULLT <Zd>.D, <Zn>.S, <Zm>.S[<imm>]
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.  
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = UInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
    integer res = element1 * element2;
    Elem[result, e, 2*esize] = res<2*esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

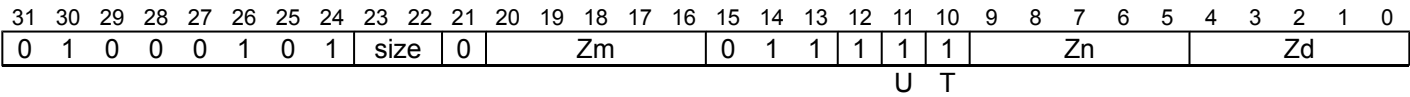
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMULLT (vectors)

Unsigned multiply long (top)

Multiply the corresponding odd-numbered unsigned elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



```
UMULLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    integer res = element1 * element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

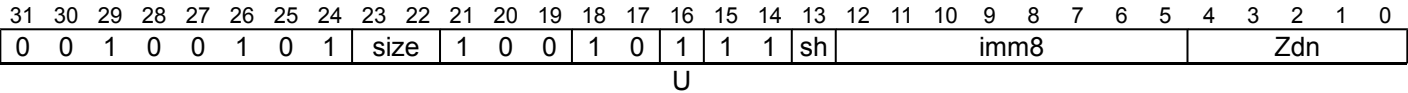
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQADD (immediate)

Unsigned saturating add immediate (unpredicated)

Unsigned saturating add of an unsigned immediate to each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. This instruction is unpredicated.  
The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".



```
UQADD <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + imm, esize, unsigned);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

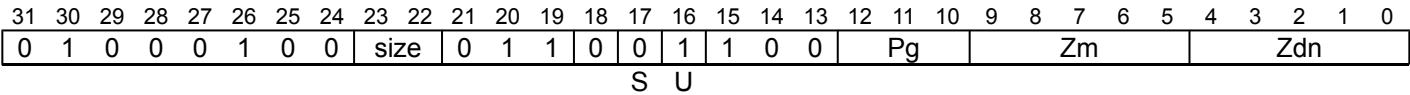
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQADD (vectors, predicated)

Unsigned saturating addition (predicated)

Add active unsigned elements of the first source vector to corresponding unsigned elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. Inactive elements in the destination vector register remain unmodified.



```
UQADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = UInt(Sat(element1 + element2, esize, unsigned));
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

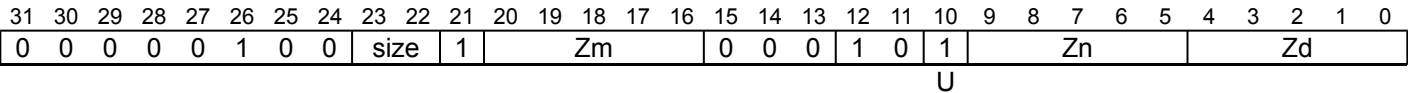
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQADD (vectors, unpredicated)

Unsigned saturating add vectors (unpredicated)

Unsigned saturating add all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. This instruction is unpredicated.



UQADD <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + element2, esize, unsigned);

Z[d, VL] = result;
```

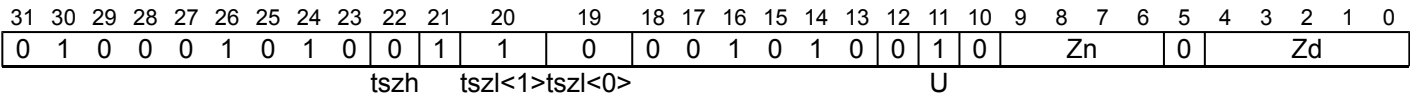


UQCVTN

Unsigned saturating extract narrow and interleave

Saturate the unsigned integer value in each element of the group of two source vectors to half the original source element width, and place the two-way interleaved results in the half-width destination elements.  
This instruction is unpredicated.

SVE2  
(FEAT\_SVE2p1)



```
UQCVTN <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;

for e = 0 to elements-1
  for i = 0 to 1
    bits(VL) operand = Z[n+i, VL];
    integer element = UInt(Elem[operand, e, 2 * esize]);
    Elem[result, 2*e + i, esize] = UnsignedSat(element, esize);

Z[d, VL] = result;
```

# UQDECB

Unsigned saturating decrement scalar by multiple of 8-bit predicate constraint element count

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

## 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	1	0	imm4			1			1	1	1	1	1	pattern				Rdn				
size<1>size<0>								sf				D U																			

UQDECB <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
constant integer ssize = 32;
```

## 64-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	1	1	imm4			1			1	1	1	1	1	pattern				Rdn				
size<1>size<0>								sf				D U																			

UQDECB <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
constant integer ssize = 64;
```

## Assembler Symbols

- <Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# UQDECD (scalar)

Unsigned saturating decrement scalar by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

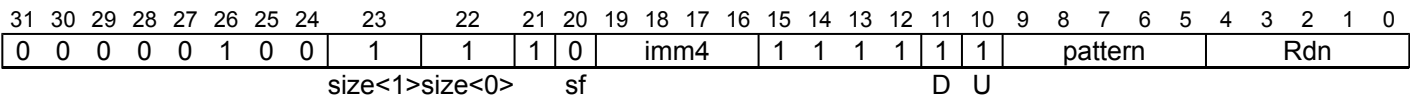
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

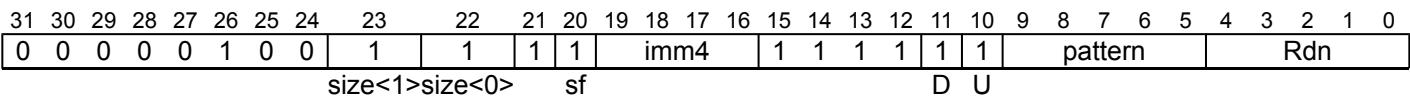
## 32-bit



```
UQDECD <Wdn>{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
constant integer ssize = 32;
```

## 64-bit



```
UQDECD <Xdn>{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
constant integer ssize = 64;
```

## Assembler Symbols

- <Wdn>
- Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Xdn>
- Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern>
- Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQDECD (vector)

Unsigned saturating decrement vector by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 64-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	1	1	0	imm4				1	1	0	0	1	1	pattern					Zdn				
size<1>size<0>																D												U			

UQDECD <Zdn>.D{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# UQDECH (scalar)

Unsigned saturating decrement scalar by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

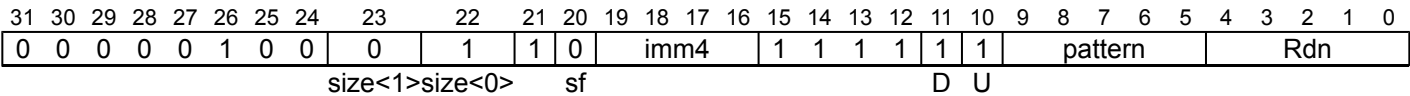
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

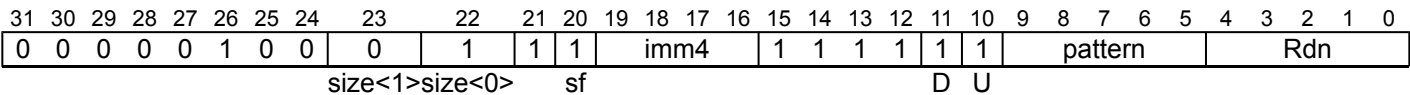
## 32-bit



```
UQDECH <Wdn>{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
constant integer ssize = 32;
```

## 64-bit



```
UQDECH <Xdn>{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
constant integer ssize = 64;
```

## Assembler Symbols

- <Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":



pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQDECH (vector)

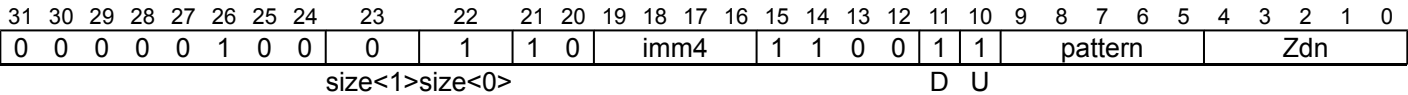
Unsigned saturating decrement vector by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 16-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



```
UQDECH <Zdn>.H{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
```

Assembler Symbols

- <Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- <pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

- <imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer elements = VL DIV esize;  
integer count = DecodePredCount(pat, esize);  
bits(VL) operand1 = Z[dn, VL];  
bits(VL) result;  
  
for e = 0 to elements-1  
    integer element1 = Int(Elem[operand1, e, esize], unsigned);  
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);  
  
Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

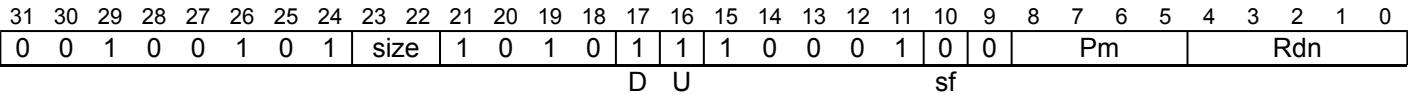
UQDECP (scalar)

Unsigned saturating decrement scalar by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

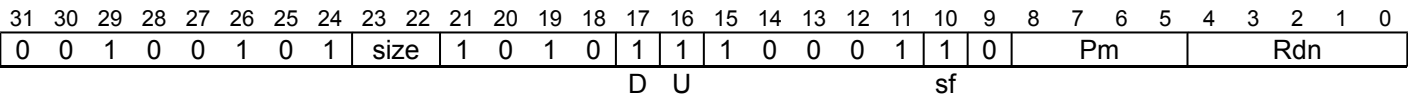
32-bit



UQDECP <Wdn>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = TRUE;
constant integer ssize = 32;
```

64-bit



UQDECP <Xdn>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = TRUE;
constant integer ssize = 64;
```

Assembler Symbols

- <Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(ssize) operand1 = X[dn, ssize];  
bits(PL) operand2 = P[m, PL];  
bits(ssize) result;  
integer count = 0;  
  
for e = 0 to elements-1  
    if ActivePredicateElement(operand2, e, esize) then  
        count = count + 1;  
  
integer element = Int(operand1, unsigned);  
(result, -) = SatQ(element - count, ssize, unsigned);  
X[dn, 64] = Extend(result, 64, unsigned);
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

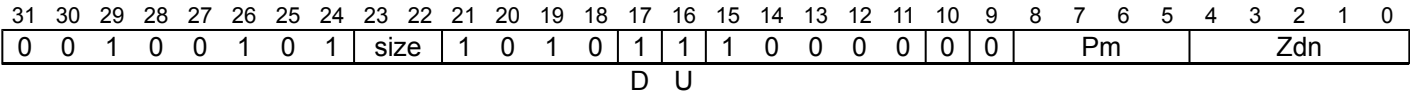
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQDECP (vector)

Unsigned saturating decrement vector by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to decrement all destination vector elements. The results are saturated to the element unsigned integer range. The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.



```
UQDECP <Zdn>.<T>, <Pm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) operand2 = P[m, PL];
bits(VL) result;
integer count = 0;

for e = 0 to elements-1
    if ActivePredicateElement(operand2, e, esize) then
        count = count + 1;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element - count, esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.

- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# UQDECW (scalar)

Unsigned saturating decrement scalar by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

## 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	0	1	0	imm4				1	1	1	1	1	1	pattern					Rdn				
size<1>size<0>								sf				D U																			

UQDECW <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
constant integer ssize = 32;
```

## 64-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	0	1	1	imm4				1	1	1	1	1	1	pattern					Rdn				
size<1>size<0>								sf				D U																			

UQDECW <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
constant integer ssize = 64;
```

## Assembler Symbols

- <Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":



pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQDECW (vector)

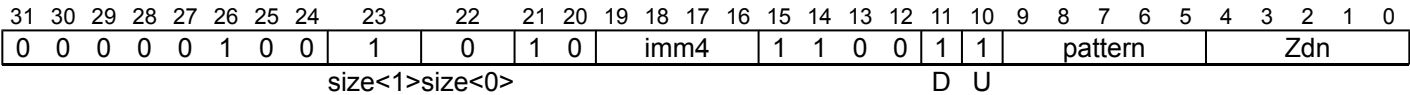
Unsigned saturating decrement vector by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 32-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



```
UQDECW <Zdn>.S{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# UQINCB

Unsigned saturating increment scalar by multiple of 8-bit predicate constraint element count

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

## 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	1	0	imm4			1	1	1	1	0	1	pattern				Rdn						
size<1>size<0>								sf				D U																			

UQINCB <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
constant integer ssize = 32;
```

## 64-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	1	1	imm4			1	1	1	1	0	1	pattern				Rdn						
size<1>size<0>								sf								D				U											

UQINCB <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
constant integer ssize = 64;
```

## Assembler Symbols

- <Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# UQINCD (scalar)

Unsigned saturating increment scalar by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

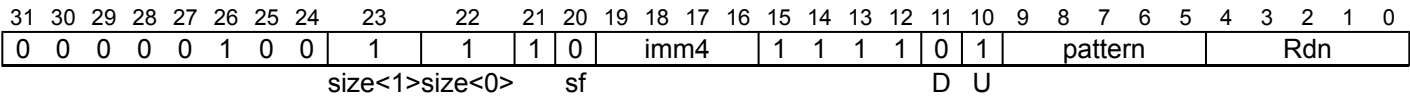
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

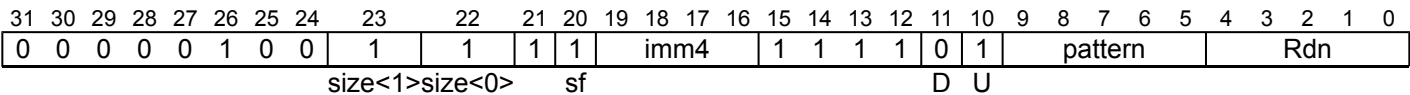
## 32-bit



```
UQINCD <Wdn>{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
constant integer ssize = 32;
```

## 64-bit



```
UQINCD <Xdn>{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
constant integer ssize = 64;
```

## Assembler Symbols

- <Wdn>

Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Xdn>

Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern>

Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQINCD (vector)

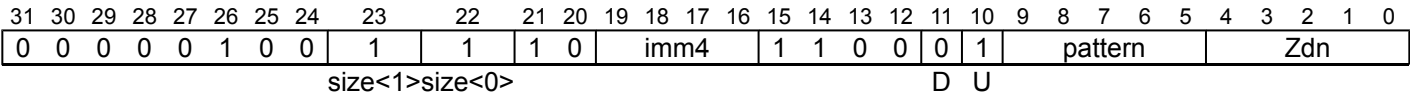
Unsigned saturating increment vector by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 64-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



```
UQINCD <Zdn>.D{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# UQINCH (scalar)

Unsigned saturating increment scalar by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

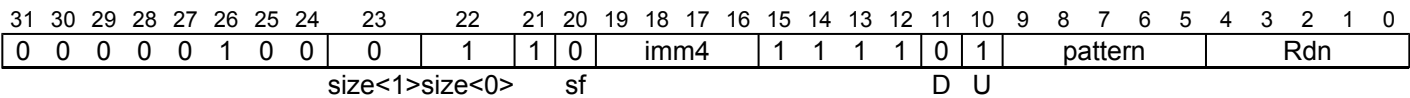
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

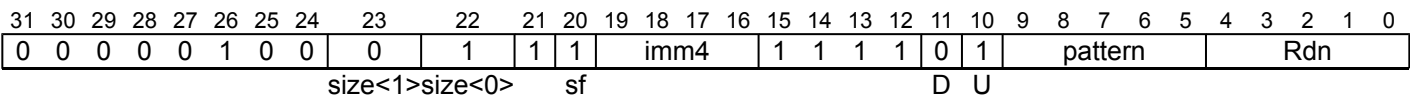
## 32-bit



UQINCH <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
constant integer ssize = 32;
```

## 64-bit



UQINCH <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
constant integer ssize = 64;
```

## Assembler Symbols

- <Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQINCH (vector)

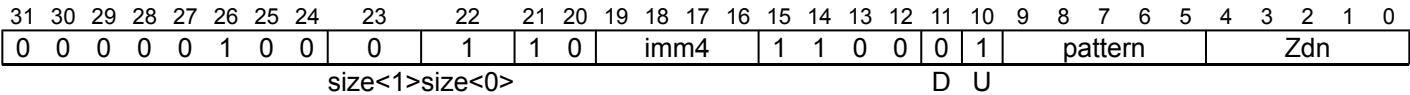
Unsigned saturating increment vector by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 16-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



```
UQINCH <Zdn>.H{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

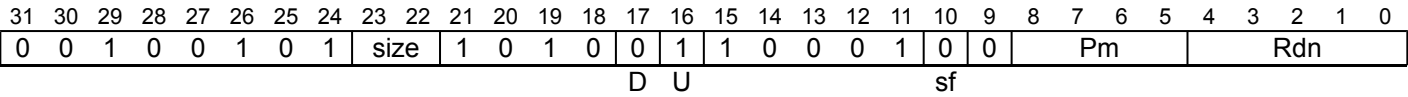
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQINCP (scalar)

Unsigned saturating increment scalar by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.  
It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

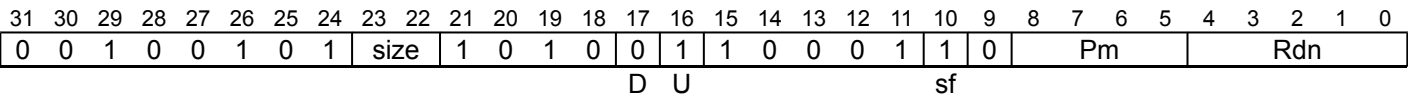
32-bit



UQINCP <Wdn>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = TRUE;
constant integer ssize = 32;
```

64-bit



UQINCP <Xdn>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = TRUE;
constant integer ssize = 64;
```

Assembler Symbols

- <Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(ssize) operand1 = X[dn, ssize];  
bits(PL) operand2 = P[m, PL];  
bits(ssize) result;  
integer count = 0;  
  
for e = 0 to elements-1  
    if ActivePredicateElement(operand2, e, esize) then  
        count = count + 1;  
  
integer element = Int(operand1, unsigned);  
(result, -) = SatQ(element + count, ssize, unsigned);  
X[dn, 64] = Extend(result, 64, unsigned);
```

## Operational information

If FEAT\_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

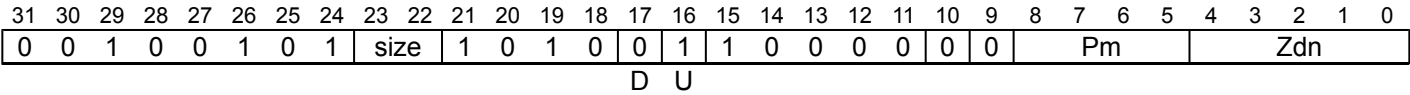
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQINCP (vector)

Unsigned saturating increment vector by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to increment all destination vector elements. The results are saturated to the element unsigned integer range.  
The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.



```
UQINCP <Zdn>.<T>, <Pm>.<T>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) operand2 = P[m, PL];
bits(VL) result;
integer count = 0;

for e = 0 to elements-1
    if ActivePredicateElement(operand2, e, esize) then
        count = count + 1;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element + count, esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.



- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# UQINCW (scalar)

Unsigned saturating increment scalar by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

## 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	0	1	0	imm4				1	1	1	1	0	1	pattern					Rdn				
size<1>size<0>								sf								D		U													

UQINCW <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
constant integer ssize = 32;
```

## 64-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	0	1	1	imm4				1	1	1	1	0	1	pattern					Rdn				
size<1>size<0>								sf				D U																			

UQINCW <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
constant integer ssize = 64;
```

## Assembler Symbols

- <Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQINCW (vector)

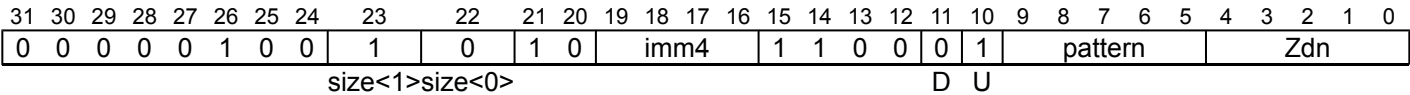
Unsigned saturating increment vector by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 32-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



```
UQINCW <Zdn>.S{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

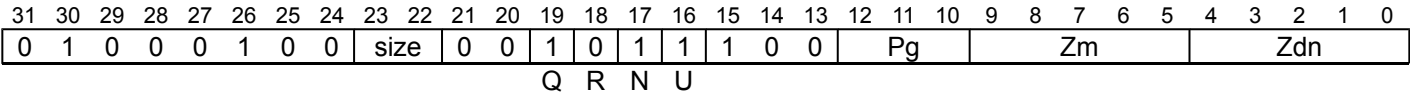
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQRSHL

Unsigned saturating rounding shift left by vector (predicated)

Shift active unsigned elements of the first source vector by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. Inactive elements in the destination vector register remain unmodified.



```
UQRSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    integer element = UInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer res;
    if shift >= 0 then
      res = element << shift;
    else
      shift = -shift;
      res = (element + (1 << (shift - 1))) >> shift;
    Elem[result, e, esize] = UnsignedSat(res, esize);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQRSHLR

Unsigned saturating rounding shift left reversed vectors (predicated)

Shift active unsigned elements of the second source vector by corresponding elements of the first source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	0	0	1	1	1	1	1	0	0	Pg	Zm				Zdn								
																Q R N U															

UQRSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand2 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer element = UInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
        integer res;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = (element + (1 << (shift - 1))) >> shift;
        Elem[result, e, esize] = UnsignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand2, e, esize];

Z[dn, VL] = result;
```



## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

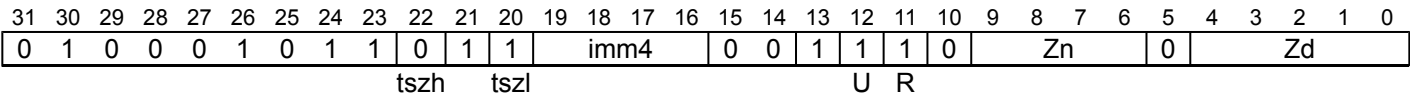
UQRSHRN

Unsigned saturating rounding shift right narrow by immediate and interleave

Shift right by an immediate value, the unsigned integer value in each element of the group of two source vectors and place the two-way interleaved rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SVE2
(FEAT\_SVE2p1)



UQRSHRN <Zd>.H, { <Zn1>.S-<Zn2>.S }, #<const>

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
integer shift = esize - UInt(imm4);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <const> Is the immediate shift amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;

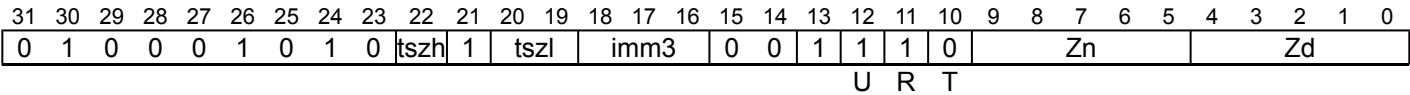
for e = 0 to elements-1
  for i = 0 to 1
    bits(VL) operand = Z[n+i, VL];
    bits(2 * esize) element = Elem[operand, e, 2 * esize];
    integer res = (UInt(element) + (1 << (shift-1))) >> shift;
    Elem[result, 2*e + i, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```

UQRSHRNB

Unsigned saturating rounding shift right narrow by immediate (bottom)

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



UQRSHRNB <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = (UInt(element) + (1 << (shift-1))) >> shift;
    Elem[result, 2*e + 0, esize] = UnsignedSat(res, esize);
    Elem[result, 2*e + 1, esize] = Zeros(esize);

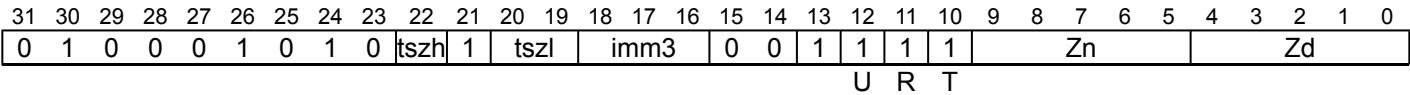
Z[d, VL] = result;
```



UQRSHRNT

Unsigned saturating rounding shift right narrow by immediate (top)

Shift each unsigned integer value in the source vector elements by an immediate value, and place the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



UQRSHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = (UInt(element) + (1 << (shift-1))) >> shift;
    Elem[result, 2*e + 1, esize] = UnsignedSat(res, esize);

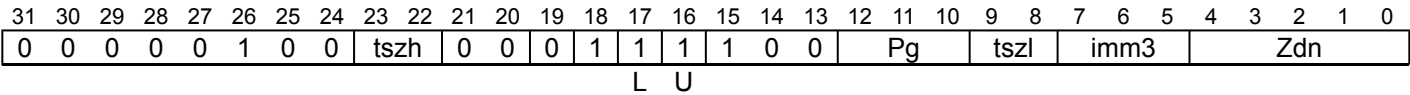
Z[d, VL] = result;
```



UQSHL (immediate)

Unsigned saturating shift left by immediate

Shift left by immediate each active unsigned element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.



```
UQSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = element1 << shift;
        Elem[result, e, esize] = UnsignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

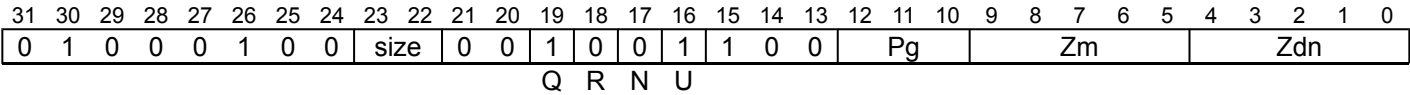
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



UQSHL (vectors)

Unsigned saturating shift left by vector (predicated)

Shift active unsigned elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. Inactive elements in the destination vector register remain unmodified.



UQSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    integer element = UInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer res;
    if shift >= 0 then
      res = element << shift;
    else
      shift = -shift;
      res = element >> shift;
    Elem[result, e, esize] = UnsignedSat(res, esize);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# UQSHLR

Unsigned saturating shift left reversed vectors (predicated)

Shift active unsigned elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. Inactive elements in the destination vector register remain unmodified.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	0	0	0	1	0	0	size	0	0	1	1	0	1	1	0	0	Pg	Zm				Zdn										
																		Q		R	N		U										

UQSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

## Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand2 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        integer element = UInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
        integer res;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = element >> shift;
        Elem[result, e, esize] = UnsignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand2, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

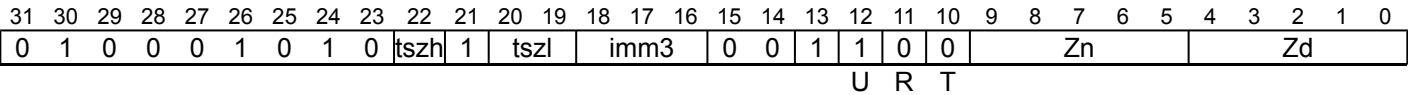
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQSHRNB

Unsigned saturating shift right narrow by immediate (bottom)

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



UQSHRNB <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = UInt(element) >> shift;
    Elem[result, 2*e + 0, esize] = UnsignedSat(res, esize);
    Elem[result, 2*e + 1, esize] = Zeros(esize);

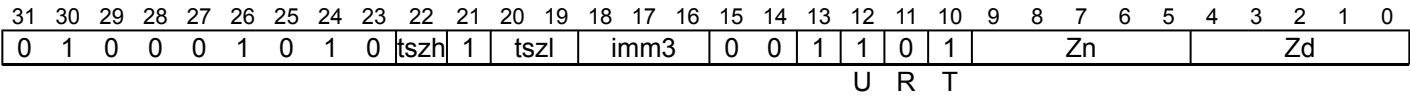
Z[d, VL] = result;
```



UQSHRNT

Unsigned saturating shift right narrow by immediate (top)

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



UQSHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = UInt(element) >> shift;
    Elem[result, 2*e + 1, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```





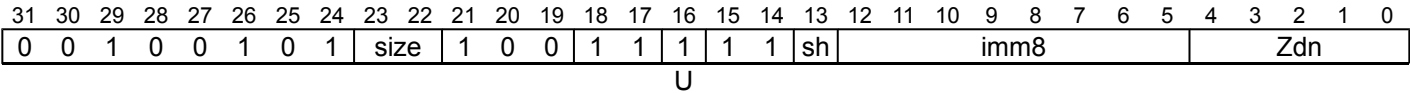
UQSUB (immediate)

Unsigned saturating subtract immediate (unpredicated)

Unsigned saturating subtract an unsigned immediate from each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".



```
UQSUB <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - imm, esize, unsigned);

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

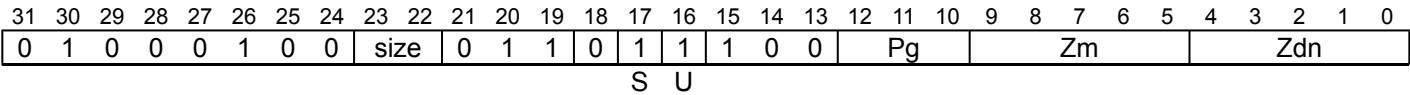
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQSUB (vectors, predicated)

Unsigned saturating subtraction (predicated)

Subtract active unsigned elements of the second source vector from corresponding unsigned elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. Inactive elements in the destination vector register remain unmodified.



UQSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = UInt(Sat(element1 - element2, esize, unsigned));
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

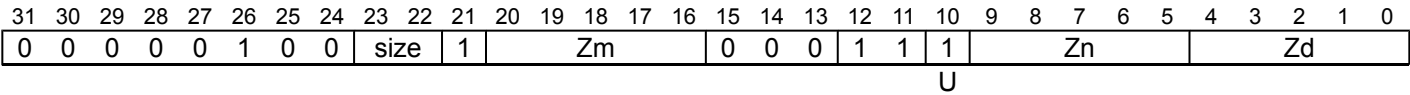
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQSUB (vectors, unpredicated)

Unsigned saturating subtract vectors (unpredicated)

Unsigned saturating subtract all elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. This instruction is unpredicated.



UQSUB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

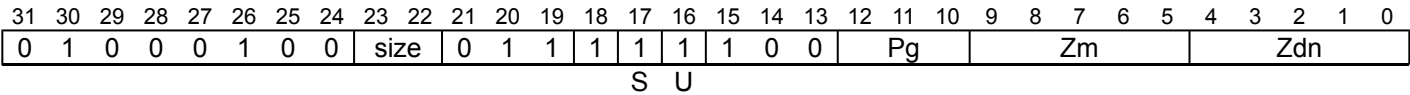
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - element2, esize, unsigned);

Z[d, VL] = result;
```

UQSUBR

Unsigned saturating subtraction reversed vectors (predicated)

Subtract active unsigned elements of the first source vector from corresponding unsigned elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. Inactive elements in the destination vector register remain unmodified.



```
UQSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = UInt(Sat(element2 - element1, esize, unsigned));
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

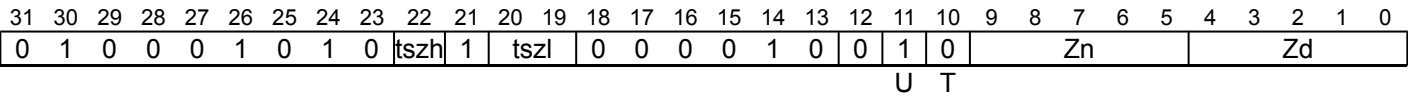
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQXTNB

Unsigned saturating extract narrow (bottom)

Saturate the unsigned integer value in each source element to half the original source element width, and place the results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero.



UQXTNB <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '001' esize = 16;
    when '010' esize = 32;
    when '100' esize = 64;
    otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	10	H
x	11	RESERVED
1	00	S
1	01	RESERVED
1	10	RESERVED

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	10	S
x	11	RESERVED
1	00	D
1	01	RESERVED
1	10	RESERVED



## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(VL) operand1 = Z[n, VL];  
bits(VL) result;  
constant integer halfesize = esize DIV 2;  
  
for e = 0 to elements-1  
    integer element1 = UInt(Elem[operand1, e, esize]);  
    bits(halfesize) res = UnsignedSat(element1, halfesize);  
    Elem[result, 2*e + 0, halfesize] = res;  
    Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);  
  
Z[d, VL] = result;
```

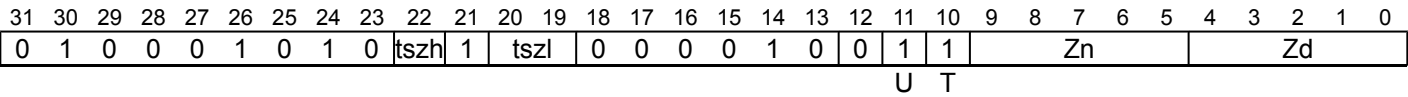
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQXTNT

Unsigned saturating extract narrow (top)

Saturate the unsigned integer value in each source element to half the original source element width, and place the results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged.



UQXTNT <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '001' esize = 16;
    when '010' esize = 32;
    when '100' esize = 64;
    otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	10	H
x	11	RESERVED
1	00	S
1	01	RESERVED
1	10	RESERVED

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	10	S
x	11	RESERVED
1	00	D
1	01	RESERVED
1	10	RESERVED

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
bits(VL) operand1 = Z[n, VL];  
bits(VL) result = Z[d, VL];  
constant integer halfesize = esize DIV 2;  
  
for e = 0 to elements-1  
    integer element1 = UInt(Elem[operand1, e, esize]);  
    bits(halfesize) res = UnsignedSat(element1, halfesize);  
    Elem[result, 2*e + 1, halfesize] = res;  
  
Z[d, VL] = result;
```

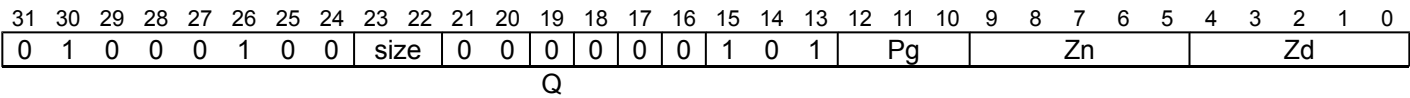
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

URECPE

Unsigned reciprocal estimate (predicated)

Find the approximate reciprocal of each active unsigned element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.



```
URECPE <Zd>.S, <Pg>/M, <Zn>.S
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size != '10' then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = UnsignedRecipEstimate(element);

Z[d, VL] = result;
```

Operational information

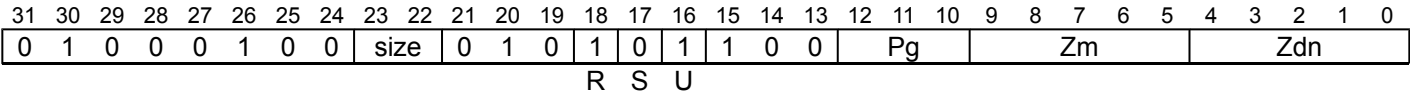
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

URHADD

Unsigned rounding halving addition

Add active unsigned elements of the first source vector to corresponding unsigned elements of the second source vector, shift right one bit, and destructively place the rounded results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



```
URHADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = (element1 + element2 + 1) >> 1;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

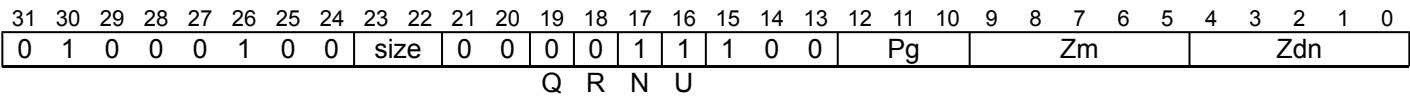
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

URSHL

Unsigned rounding shift left by vector (predicated)

Shift active unsigned elements of the first source vector by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.



```
URSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    integer element = UInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer res;
    if shift >= 0 then
      res = element << shift;
    else
      shift = -shift;
      res = (element + (1 << (shift - 1))) >> shift;
    Elem[result, e, esize] = res<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

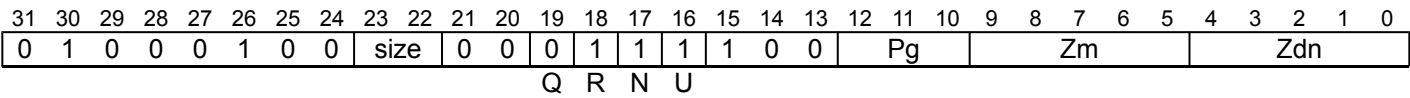
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



URSHLR

Unsigned rounding shift left reversed vectors (predicated)

Shift active unsigned elements of the second source vector by corresponding elements of the first source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.



```
URSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand2 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
  if ActivePredicateElement(mask, e, esize) then
    integer element = UInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer res;
    if shift >= 0 then
      res = element << shift;
    else
      shift = -shift;
      res = (element + (1 << (shift - 1))) >> shift;
    Elem[result, e, esize] = res<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand2, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

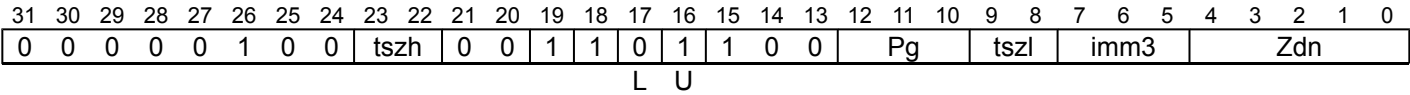
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

URSHR

Unsigned rounding shift right by immediate

Shift right by immediate each active unsigned element of the source vector, and destructively place the rounded results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.



```
URSHR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

- <Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- <T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    if ActivePredicateElement(mask, e, esize) then
        integer res = (element1 + (1 << (shift - 1))) >> shift;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

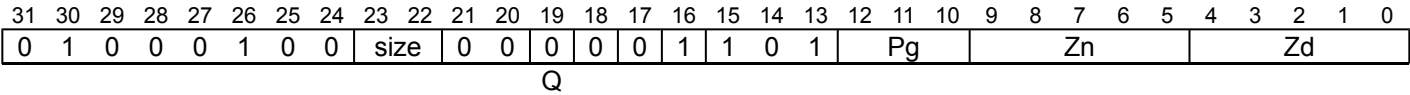
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# URSQRTE

Unsigned reciprocal square root estimate (predicated)

Find the approximate reciprocal square root of each active unsigned element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.



```
URSQRTE <Zd>.S, <Pg>/M, <Zn>.S
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size != '10' then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

## Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = UnsignedRSqrtEstimate(element);

Z[d, VL] = result;
```

## Operational information

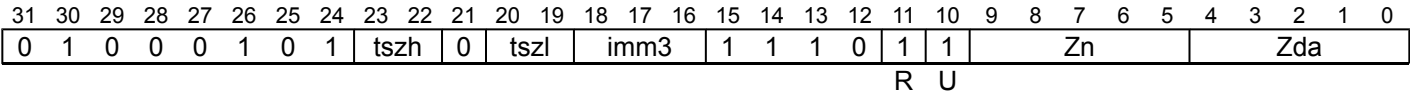
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

URSRA

Unsigned rounding shift right and accumulate (immediate)

Shift right by immediate each unsigned element of the source vector, inserting zeroes, and add the rounded intermediate result destructively to the corresponding elements of the addend vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



URSRA <Zda>.<T>, <Zn>.<T>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer da = UInt(Zda);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element = (UInt(Elem[operand1, e, esize]) + (1 << (shift - 1))) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



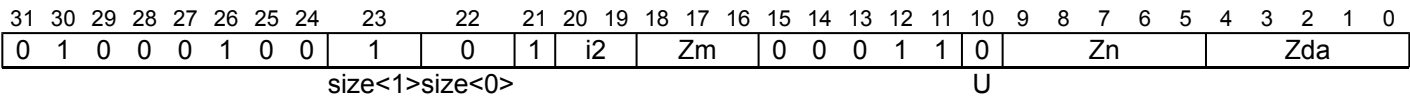
# USDOT (indexed)

Unsigned by signed integer indexed dot product

The unsigned by signed integer indexed dot product instruction computes the dot product of a group of four unsigned 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four signed 8-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated. ID\_AA64ZFR0\_EL1.I8MM indicates whether this instruction is implemented.

## SVE (FEAT\_I8MM)



USDOT <Zda>.S, <Zn>.B, <Zm>.B[<imm>]

```
if (!HaveSVE() && !HaveSME()) || !HaveInt8MatMulExt() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

## Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```



## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## USDOT (vectors)

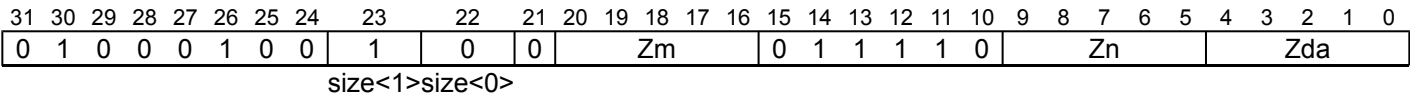
Unsigned by signed integer dot product

The unsigned by signed integer dot product instruction computes the dot product of a group of four unsigned 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four signed 8-bit integer values in the corresponding 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.I8MM indicates whether this instruction is implemented.

### SVE (FEAT\_I8MM)



USDOT <Zda>.S, <Zn>.B, <Zm>.B

```
if (!HaveSVE() && !HaveSME()) || !HaveInt8MatMulExt() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

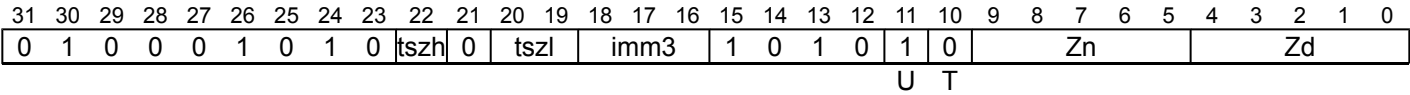
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USHLLB

Unsigned shift left long by immediate (bottom)

Shift left by immediate each even-numbered unsigned element of the source vector, and place the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.



```
USHLLB <Zd>.<T>, <Zn>.<Tb>, #<const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand, 2*e + 0, esize];
    integer shifted_value = UInt(element) << shift;
    Elem[result, e, 2*esize] = shifted_value<2*esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

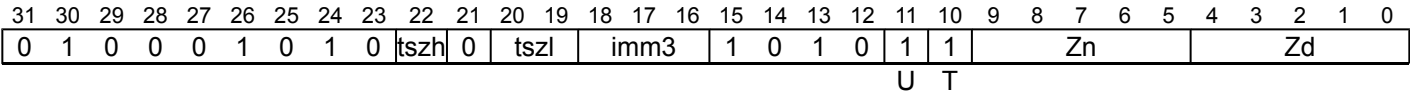
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USHLLT

Unsigned shift left long by immediate (top)

Shift left by immediate each odd-numbered unsigned element of the source vector, and place the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.



```
USHLLT <Zd>.<T>, <Zn>.<Tb>, #<const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
if tsize == '000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<Tb>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand, 2*e + 1, esize];
    integer shifted_value = UInt(element) << shift;
    Elem[result, e, 2*esize] = shifted_value<2*esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USMMLA

Unsigned by signed integer matrix multiply-accumulate

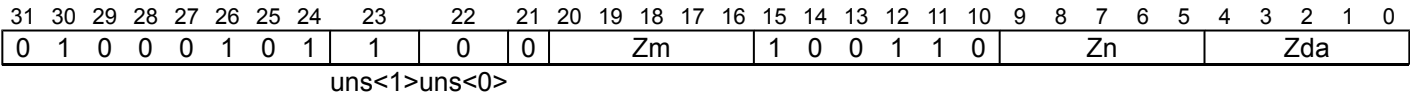
The unsigned by signed integer matrix multiply-accumulate instruction multiplies the 2×8 matrix of unsigned 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of signed 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

This instruction is unpredicated.

ID\_AA64ZFR0\_EL1.I8MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

SVE  
(FEAT\_I8MM)



```
USMMLA <Zda>.S, <Zn>.B, <Zm>.B
```

```
if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;
```

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result = Zeros(VL);
bits(128) op1, op2;
bits(128) res, addend;

for s = 0 to segments-1
    op1 = Elem[operand1, s, 128];
    op2 = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];
    res = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
    Elem[result, s, 128] = res;

Z[da, VL] = result;
```



## Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

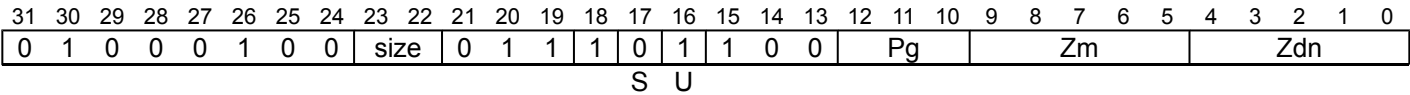
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USQADD

Unsigned saturating addition of signed value

Add active signed elements of the source vector to the corresponding unsigned elements of the addend vector, and destructively place the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. Inactive elements in the destination vector register remain unmodified.



```
USQADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = UnsignedSat(UInt(element1) + SInt(element2), esize);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

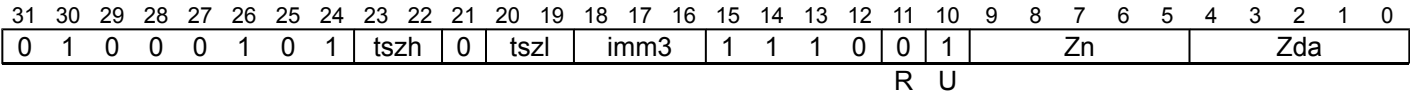
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USRA

Unsigned shift right and accumulate (immediate)

Shift right by immediate each unsigned element of the source vector, inserting zeroes, and add the truncated intermediate result destructively to the corresponding elements of the addend vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



```
USRA <Zda>.<T>, <Zn>.<T>, #<const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn);
integer da = UInt(Zda);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element = UInt(Elem[operand1, e, esize]) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

Z[da, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

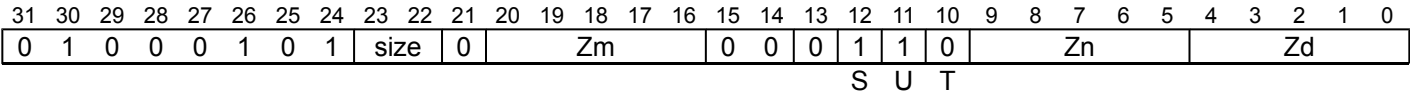
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USUBLB

Unsigned subtract long (bottom)

Subtract the even-numbered unsigned elements of the second source vector from the corresponding unsigned elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



```
USUBLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 0;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

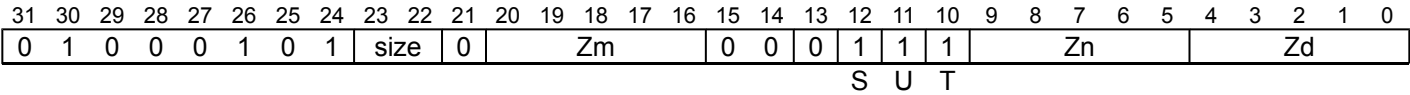
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USUBLT

Unsigned subtract long (top)

Subtract the odd-numbered unsigned elements of the second source vector from the corresponding unsigned elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



```
USUBLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 1;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```



## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

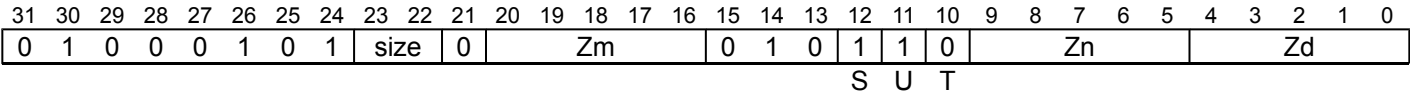
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USUBWB

Unsigned subtract wide (bottom)

Subtract the even-numbered unsigned elements of the second source vector from the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.



```
USUBWB <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    Elem[result, e, esize] = (element1 - element2)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

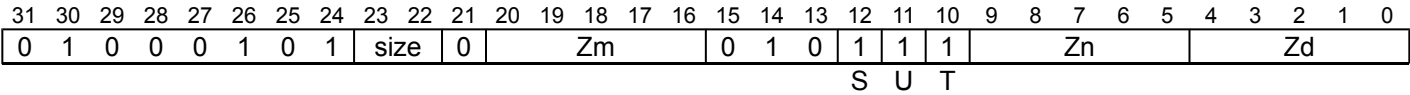
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USUBWT

Unsigned subtract wide (top)

Subtract the odd-numbered unsigned elements of the second source vector from the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated. This instruction is unpredicated.



```
USUBWT <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    Elem[result, e, esize] = (element1 - element2)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

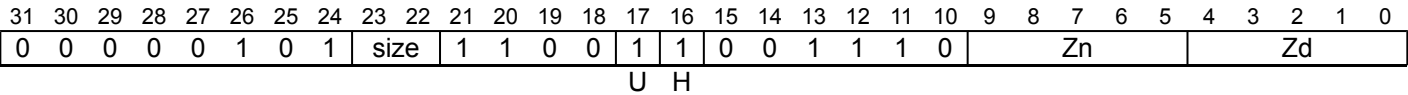
# UUNPKHI, UUNPKLO

Unsigned unpack and extend half of vector

Unpack elements from the lowest or highest half of the source vector and then zero-extend them to place in elements of twice their size within the destination vector. This instruction is unpredicated.

It has encodings from 2 classes: [High half](#) and [Low half](#)

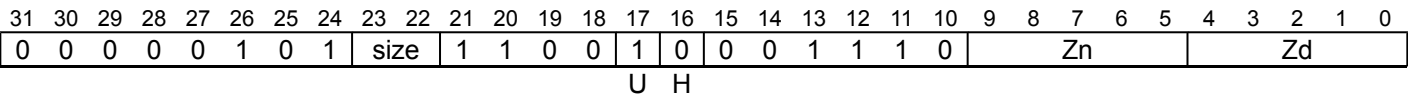
## High half



UUNPKHI <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
boolean hi = TRUE;
```

## Low half



UUNPKLO <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
boolean hi = FALSE;
```

## Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

## Operation

```
CheckSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
constant integer elements = VL DIV esize;  
constant integer hsize = esize DIV 2;  
bits(VL) operand = Z[n, VL];  
bits(VL) result;  
  
for e = 0 to elements-1  
    bits(hsize) element = if hi then Elem[operand, e + elements, hsize] else Elem[operand, e, hsize];  
    Elem[result, e, esize] = Extend(element, esize, unsigned);  
  
Z[d, VL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

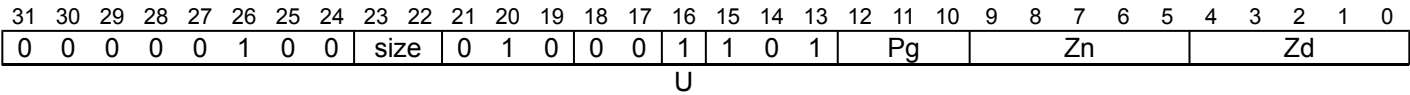
UXTB, UXTH, UXTW

Unsigned byte / halfword / word extend (predicated)

Zero-extend the least-significant sub-element of each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

It has encodings from 3 classes: [Byte](#) , [Halfword](#) and [Word](#)

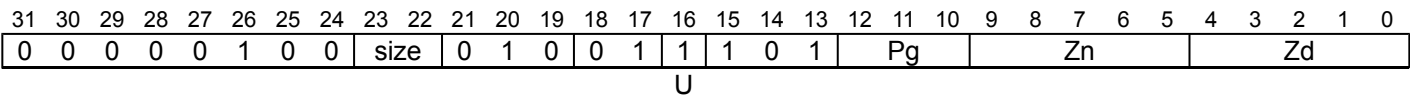
Byte



UXTB <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 8;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
```

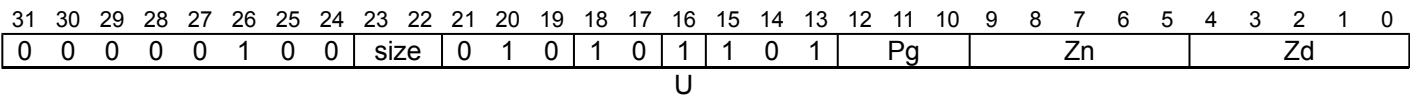
Halfword



UXTH <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
```

Word



UXTW <Zd>.D, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size != '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
```



Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <T> For the byte variant: is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

For the halfword variant: is the size specifier, encoded in "size<0>":

size<0>	<T>
0	S
1	D

- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActivePredicateElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = Extend(element<s_ensure-1:0>, esize, unsigned);

Z[d, VL] = result;
```

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

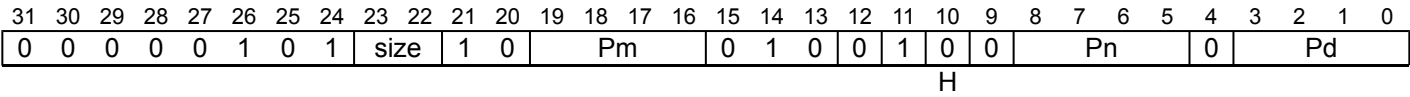
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

UZP1, UZP2 (predicates)

Concatenate even or odd elements from two predicates

Concatenate adjacent even or odd-numbered elements from the first and second source predicates and place in elements of the destination predicate. This instruction is unpredicated.  
It has encodings from 2 classes: [Even](#) and [Odd](#)

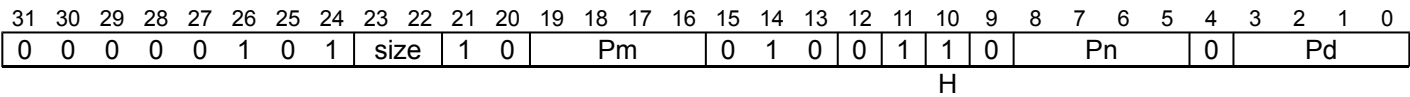
Even



UZP1 <Pd>.<T>, <Pn>.<T>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 0;
```

Odd



UZP2 <Pd>.<T>, <Pn>.<T>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 1;
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D
- <Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (esize * 2);
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for p = 0 to pairs - 1
    Elem[result, p, esize DIV 8] = Elem[operand1, 2*p+part, esize DIV 8];

for p = 0 to pairs - 1
    Elem[result, pairs+p, esize DIV 8] = Elem[operand2, 2*p+part, esize DIV 8];

P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UZP1, UZP2 (vectors)

Concatenate even or odd elements from two vectors

Concatenate adjacent even or odd-numbered elements from the first and second source vectors and place in elements of the destination vector. This instruction is unpredicated.

Note: UZP1 is equivalent to truncating and packing each element from two source vectors into a single destination vector with elements of half the size.

The 128-bit element variant requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero. ID\_AA64ZFR0\_EL1.F64MM indicates whether the 128-bit element variant is implemented. The 128-bit element variant is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

It has encodings from 4 classes: [Even](#) , [Even \(quadwords\)](#) , [Odd](#) and [Odd \(quadwords\)](#)

### Even

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1				Zm			0	1	1	0	1	0				Zn				Zd		
																H															

**UZP1** [<Zd>.<T>](#), [<Zn>.<T>](#), [<Zm>.<T>](#)

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

### Even (quadwords) (FEAT\_F64MM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	1	0	1			Zm			0	0	0	0	1	0				Zn				Zd		
																H															

**UZP1** [<Zd>.Q](#), [<Zn>.Q](#), [<Zm>.Q](#)

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

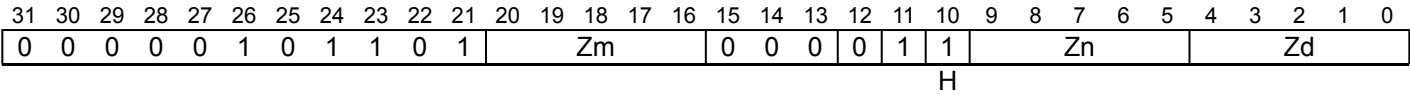
### Odd

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1				Zm			0	1	1	0	1	1				Zn				Zd		
																H															

**UZP2** [<Zd>.<T>](#), [<Zn>.<T>](#), [<Zm>.<T>](#)

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```

Odd (quadwords)  
(FEAT\_F64MM)



UZP2 <Zd>.Q, <Zn>.Q, <Zm>.Q

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
if VL < esize * 2 then UNDEFINED;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Zeros(VL);

for p = 0 to pairs - 1
    Elem[result, p, esize] = Elem[operand1, 2*p+part, esize];

for p = 0 to pairs - 1
    Elem[result, pairs+p, esize] = Elem[operand2, 2*p+part, esize];

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

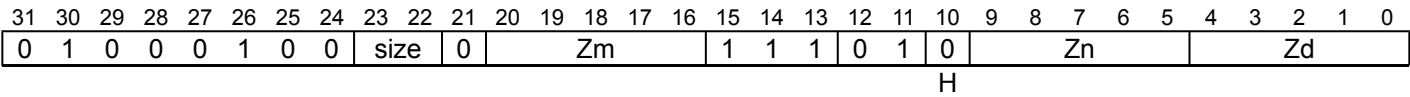
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

UZIPQ1

Concatenate even elements within each pair of quadword vector segments

Concatenate adjacent even-numbered elements from the corresponding 128-bit vector segments of the first and second source vectors and place in elements of the corresponding destination vector segment. This instruction is unpredicated.

SVE2  
(FEAT\_SVE2p1)



```
UZIPQ1 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elements = 128 DIV esize;
constant integer pairs = elements DIV 2;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for s = 0 to segments-1
  for p = 0 to pairs-1
    Elem[result, s * elements + p, esize] = Elem[operand1, s * elements + 2 * p + part, esize];

  for p = 0 to pairs-1
    Elem[result, s * elements + pairs + p, esize] = Elem[operand2, s * elements + 2 * p + part, esize];

Z[d, VL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

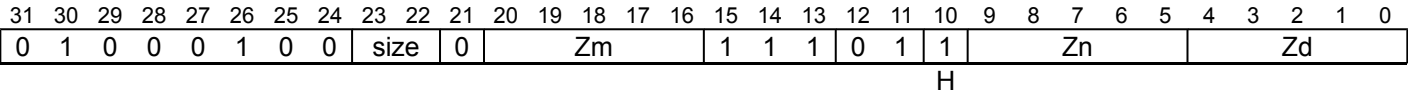
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UZIPQ2

Concatenate odd elements within each pair of quadword vector segments

Concatenate adjacent odd-numbered elements from the corresponding 128-bit vector segments of the first and second source vectors and place in elements of the corresponding destination vector segment. This instruction is unpredicated.

SVE2  
(FEAT\_SVE2p1)



```
UZIPQ2 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elements = 128 DIV esize;
constant integer pairs = elements DIV 2;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for s = 0 to segments-1
  for p = 0 to pairs-1
    Elem[result, s * elements + p, esize] = Elem[operand1, s * elements + 2 * p + part, esize];

  for p = 0 to pairs-1
    Elem[result, s * elements + pairs + p, esize] = Elem[operand2, s * elements + 2 * p + part, esize];

Z[d, VL] = result;
```



## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILEGE (predicate as counter)

While decrementing signed scalar greater than or equal to scalar (predicate-as-counter)

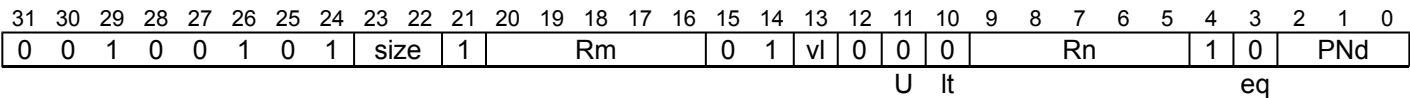
Generate a predicate for a group of two or four vectors that starting from the highest numbered element of the group is true while the decrementing value of the first, signed scalar operand is greater than or equal to the second scalar operand and false thereafter down to the lowest numbered element of the group.

If the second scalar operand is equal to the minimum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2
(FEAT\_SVE2p1)



WHILEGE <PNd>.<T>, <Xn>, <Xm>, <vl>

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt('1':PNd);
boolean unsigned = FALSE;
boolean invert = TRUE;
SVEComp op = Cmp_GE;
integer width = 2 << UInt(vl);
```

Assembler Symbols

<PNd> Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

<vl> Is the vl specifier, encoded in "vl":

vl	<vl>
0	VLx2
1	VLx4

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = width * (VL DIV esize);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
integer count = 0;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    if last then count = count + 1;
    operand1 = operand1 - 1;

result = EncodePredCount(esize, elements, count, invert, PL);
PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert);
P[d, PL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILEGE (predicate pair)

While decrementing signed scalar greater than or equal to scalar (pair of predicates)

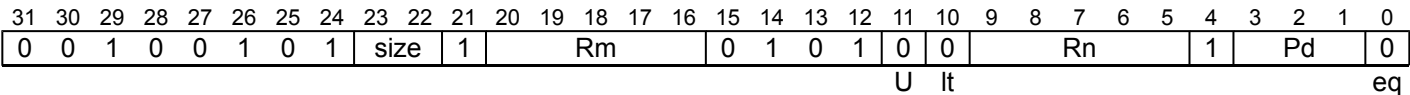
Generate a pair of predicates that starting from the highest numbered element of the pair is true while the decrementing value of the first, signed scalar operand is greater than or equal to the second scalar operand and false thereafter down to the lowest numbered element of the pair.

If the second scalar operand is equal to the minimum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2  
(FEAT\_SVE2p1)



```
WHILEGE { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d0 = UInt(Pd:'0');
integer d1 = UInt(Pd:'1');
boolean unsigned = FALSE;
SVEComp op = Cmp_GE;
```

Assembler Symbols

<Pd1> Is the name of the first destination scalable predicate register, encoded as "Pd" times 2.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.

<Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL*2) mask = Ones(PL*2);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL*2) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = (elements*2)-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d0, PL] = result<PL-1:0>;
P[d1, PL] = result<PL*2-1:PL>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILEGE (predicate)

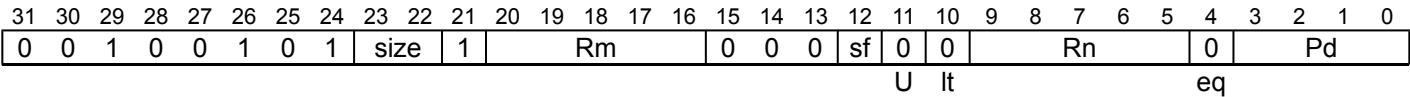
While decrementing signed scalar greater than or equal to scalar

Generate a predicate that starting from the highest numbered element is true while the decrementing value of the first, signed scalar operand is greater than or equal to the second scalar operand and false thereafter down to the lowest numbered element.

If the second scalar operand is equal to the minimum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



```
WHILEGE <Pd>.<T>, <R><n>, <R><m>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = FALSE;
SVEComp op = Cmp_GE;
```

Assembler Symbols

- <Pd>

Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D
- <R>

Is a width specifier, encoded in "sf":

sf	<R>
0	W
1	X
- <n>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.
- <m>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILEGT (predicate as counter)

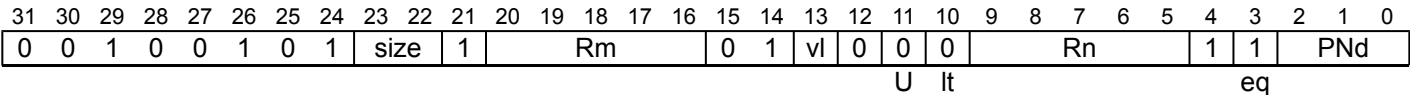
While decrementing signed scalar greater than scalar (predicate-as-counter)

Generate a predicate for a group of two or four vectors that starting from the highest numbered element of the group is true while the decrementing value of the first, signed scalar operand is greater than the second scalar operand and false thereafter down to the lowest numbered element of the group.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2  
(FEAT\_SVE2p1)



```
WHILEGT <PNd>.<T>, <Xn>, <Xm>, <vl>
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt('1':PNd);
boolean unsigned = FALSE;
boolean invert = TRUE;
SVEComp op = Cmp_GT;
integer width = 2 << UInt(vl);
```

Assembler Symbols

- <PNd> Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNd" field.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D
- <Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.
- <vl> Is the vl specifier, encoded in "vl":

vl	<vl>
0	VLx2
1	VLx4



## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = width * (VL DIV esize);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
integer count = 0;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    if last then count = count + 1;
    operand1 = operand1 - 1;

result = EncodePredCount(esize, elements, count, invert, PL);
PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert);
P[d, PL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILEGT (predicate pair)

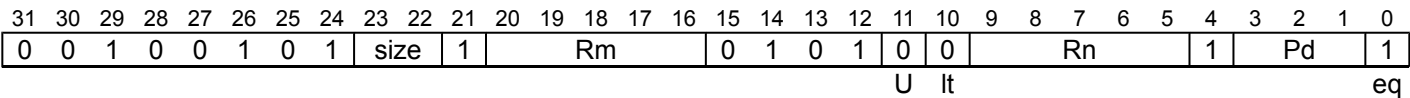
While decrementing signed scalar greater than scalar (pair of predicates)

Generate a pair of predicates that starting from the highest numbered element of the pair is true while the decrementing value of the first, signed scalar operand is greater than the second scalar operand and false thereafter down to the lowest numbered element of the pair.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2
(FEAT\_SVE2p1)



WHILEGT { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d0 = UInt(Pd:'0');
integer d1 = UInt(Pd:'1');
boolean unsigned = FALSE;
SVEComp op = Comp_GT;
```

Assembler Symbols

<Pd1> Is the name of the first destination scalable predicate register, encoded as "Pd" times 2.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.

<Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL*2) mask = Ones(PL*2);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL*2) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = (elements*2)-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d0, PL] = result<PL-1:0>;
P[d1, PL] = result<PL*2-1:PL>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

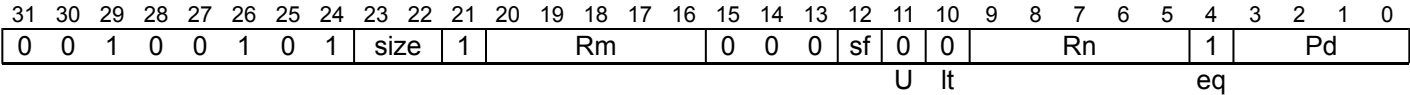
# WHILEGT (predicate)

While decrementing signed scalar greater than scalar

Generate a predicate that starting from the highest numbered element is true while the decrementing value of the first, signed scalar operand is greater than the second scalar operand and false thereafter down to the lowest numbered element.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



```
WHILEGT <Pd>.<T>, <R><n>, <R><m>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = FALSE;
SVEComp op = Cmp_GT;
```

## Assembler Symbols

- <Pd>

Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D
- <R>

Is a width specifier, encoded in "sf":

sf	<R>
0	W
1	X
- <n>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.
- <m>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILEHI (predicate as counter)

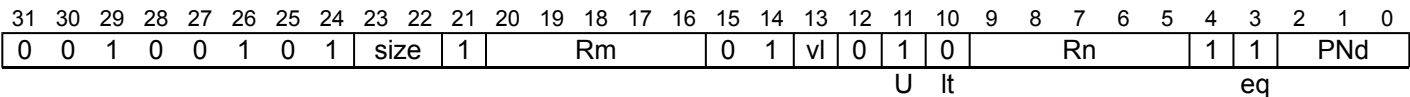
While decrementing unsigned scalar higher than scalar (predicate-as-counter)

Generate a predicate for a group of two or four vectors that starting from the highest numbered element of the group is true while the decrementing value of the first, unsigned scalar operand is higher than the second scalar operand and false thereafter down to the lowest numbered element of the group.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2
(FEAT\_SVE2p1)



WHILEHI <PNd>.<T>, <Xn>, <Xm>, <vl>

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt('1':PNd);
boolean unsigned = TRUE;
boolean invert = TRUE;
SVEComp op = Cmp_GT;
integer width = 2 << UInt(vl);
```

Assembler Symbols

- <PNd> Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNd" field.
- <T> Is the size specifier, encoded in "size":
- Table with size (00, 01, 10, 11) and <T> (B, H, S, D)
- <Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.
- <vl> Is the vl specifier, encoded in "vl":
- Table with vl (0, 1) and <vl> (VLx2, VLx4)

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = width * (VL DIV esize);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
integer count = 0;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    if last then count = count + 1;
    operand1 = operand1 - 1;

result = EncodePredCount(esize, elements, count, invert, PL);
PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert);
P[d, PL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILEHI (predicate pair)

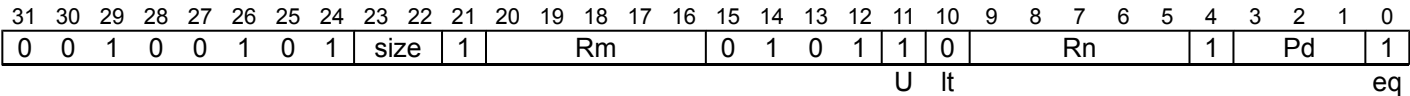
While decrementing unsigned scalar higher than scalar (pair of predicates)

Generate a pair of predicates that starting from the highest numbered element of the pair is true while the decrementing value of the first, unsigned scalar operand is higher than the second scalar operand and false thereafter down to the lowest numbered element of the pair.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2  
(FEAT\_SVE2p1)



```
WHILEHI { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d0 = UInt(Pd:'0');
integer d1 = UInt(Pd:'1');
boolean unsigned = TRUE;
SVEComp op = Cmp_GT;
```

Assembler Symbols

<Pd1> Is the name of the first destination scalable predicate register, encoded as "Pd" times 2.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.

<Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL*2) mask = Ones(PL*2);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL*2) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = (elements*2)-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d0, PL] = result<PL-1:0>;
P[d1, PL] = result<PL*2-1:PL>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

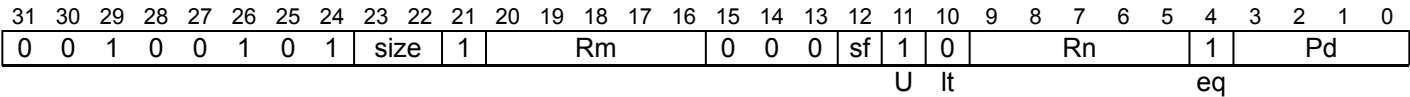
WHILEHI (predicate)

While decrementing unsigned scalar higher than scalar

Generate a predicate that starting from the highest numbered element is true while the decrementing value of the first, unsigned scalar operand is higher than the second scalar operand and false thereafter down to the lowest numbered element.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



```
WHILEHI <Pd>.<T>, <R><n>, <R><m>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = TRUE;
SVEComp op = Cmp_GT;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in "sf":

sf	<R>
0	W
1	X

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILEHS (predicate as counter)

While decrementing unsigned scalar higher or same as scalar (predicate-as-counter)

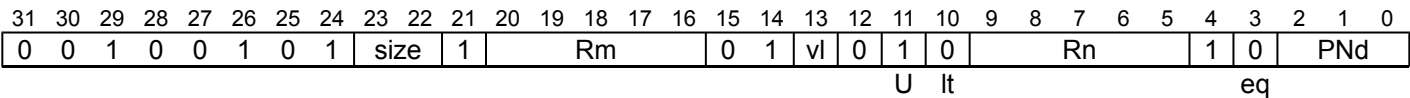
Generate a predicate for a group of two or four vectors that starting from the highest numbered element of the group is true while the decrementing value of the first, unsigned scalar operand is higher or same as the second scalar operand and false thereafter down to the lowest numbered element of the group.

If the second scalar operand is equal to the minimum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2
(FEAT\_SVE2p1)



WHILEHS <PNd>.<T>, <Xn>, <Xm>, <vl>

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt('1':PNd);
boolean unsigned = TRUE;
boolean invert = TRUE;
SVEComp op = Cmp_GE;
integer width = 2 << UInt(vl);
```

Assembler Symbols

<PNd> Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNd" field.

<T> Is the size specifier, encoded in "size":

Table with 2 columns: size, <T>. Rows: 00 B, 01 H, 10 S, 11 D.

<Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

<vl> Is the vl specifier, encoded in "vl":

Table with 2 columns: vl, <vl>. Rows: 0 VLx2, 1 VLx4.

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = width * (VL DIV esize);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
integer count = 0;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    if last then count = count + 1;
    operand1 = operand1 - 1;

result = EncodePredCount(esize, elements, count, invert, PL);
PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert);
P[d, PL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILEHS (predicate pair)

While decrementing unsigned scalar higher or same as scalar (pair of predicates)

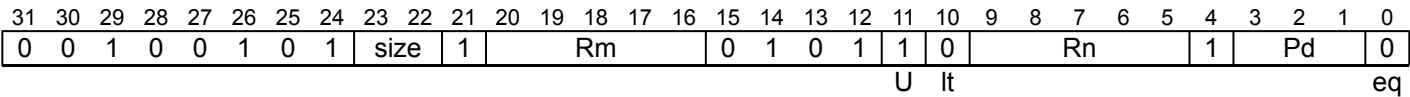
Generate a pair of predicates that starting from the highest numbered element of the pair is true while the decrementing value of the first, unsigned scalar operand is higher or same as the second scalar operand and false thereafter down to the lowest numbered element of the pair.

If the second scalar operand is equal to the minimum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2  
(FEAT\_SVE2p1)



```
WHILEHS { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d0 = UInt(Pd:'0');
integer d1 = UInt(Pd:'1');
boolean unsigned = TRUE;
SVEComp op = Cmp_GE;
```

Assembler Symbols

- <Pd1> Is the name of the first destination scalable predicate register, encoded as "Pd" times 2.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D
- <Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.
- <Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL*2) mask = Ones(PL*2);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL*2) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = (elements*2)-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d0, PL] = result<PL-1:0>;
P[d1, PL] = result<PL*2-1:PL>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## WHILEHS (predicate)

While decrementing unsigned scalar higher or same as scalar

Generate a predicate that starting from the highest numbered element is true while the decrementing value of the first, unsigned scalar operand is higher or same as the second scalar operand and false thereafter down to the lowest numbered element.

If the second scalar operand is equal to the minimum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	Rm			0	0	0	sf	1	0	Rn			0	Pd								
U lt																					eq										

WHILEHS <Pd>.<T>, <R><n>, <R><m>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = TRUE;
SVEComp op = Cmp_GE;
```

### Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in "sf":

sf	<R>
0	W
1	X

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILELE (predicate as counter)

While incrementing signed scalar less than or equal to scalar (predicate-as-counter)

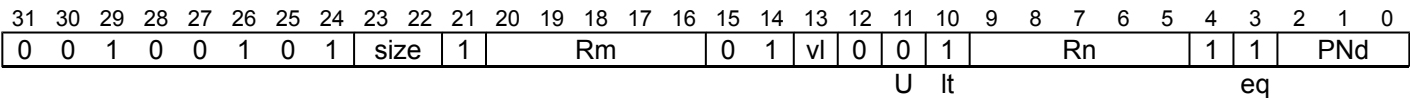
Generate a predicate for a group of two or four vectors that starting from the lowest numbered element of the group is true while the incrementing value of the first, signed scalar operand is less than or equal to the second scalar operand and false thereafter up to the highest numbered element of the group.

If the second scalar operand is equal to the maximum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2
(FEAT\_SVE2p1)



WHILELE <PNd>.<T>, <Xn>, <Xm>, <vl>

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt('1':PNd);
boolean unsigned = FALSE;
boolean invert = FALSE;
SVEComp op = Cmp_LE;
integer width = 2 << UInt(vl);
```

Assembler Symbols

<PNd> Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

<vl> Is the vl specifier, encoded in "vl":

vl	<vl>
0	VLx2
1	VLx4

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = width * (VL DIV esize);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
integer count = 0;

for e = 0 to elements-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    if last then count = count + 1;
    operand1 = operand1 + 1;

result = EncodePredCount(esize, elements, count, invert, PL);
PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert);
P[d, PL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILELE (predicate pair)

While incrementing signed scalar less than or equal to scalar (pair of predicates)

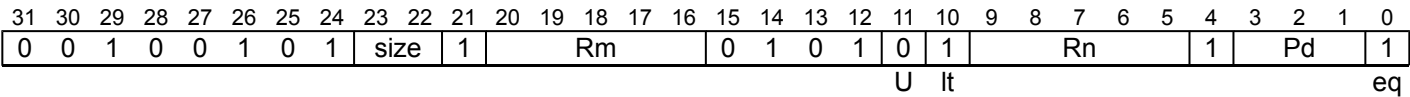
Generate a pair of predicates that starting from the lowest numbered element of the pair is true while the incrementing value of the first, signed scalar operand is less than or equal to the second scalar operand and false thereafter up to the highest numbered element of the pair.

If the second scalar operand is equal to the maximum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2  
(FEAT\_SVE2p1)



```
WHILELE { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d0 = UInt(Pd:'0');
integer d1 = UInt(Pd:'1');
boolean unsigned = FALSE;
SVEComp op = Cmp_LE;
```

Assembler Symbols

- <Pd1> Is the name of the first destination scalable predicate register, encoded as "Pd" times 2.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D
- <Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.
- <Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL*2) mask = Ones(PL*2);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL*2) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = 0 to (elements*2)-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d0, PL] = result<PL-1:0>;
P[d1, PL] = result<PL*2-1:PL>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILELE (predicate)

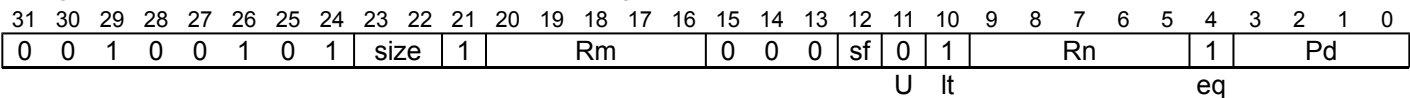
While incrementing signed scalar less than or equal to scalar

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, signed scalar operand is less than or equal to the second scalar operand and false thereafter up to the highest numbered element.

If the second scalar operand is equal to the maximum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



```
WHILELE <Pd>.<T>, <R><n>, <R><m>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = FALSE;
SVEComp op = Comp_LE;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in "sf":

sf	<R>
0	W
1	X

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILELO (predicate as counter)

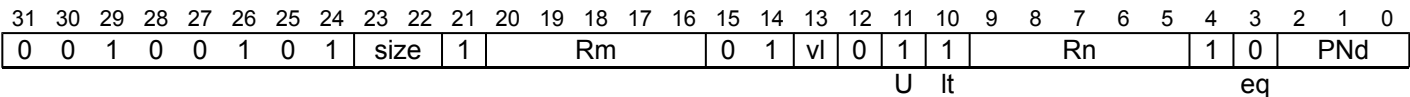
While incrementing unsigned scalar lower than scalar (predicate-as-counter)

Generate a predicate for a group of two or four vectors that starting from the lowest numbered element of the group is true while the incrementing value of the first, unsigned scalar operand is lower than the second scalar operand and false thereafter up to the highest numbered element of the group.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2  
(FEAT\_SVE2p1)



WHILELO <PNd>.<T>, <Xn>, <Xm>, <vl>

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt('1':PNd);
boolean unsigned = TRUE;
boolean invert = FALSE;
SVEComp op = Cmp_LT;
integer width = 2 << UInt(vl);
```

Assembler Symbols

<PNd> Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

<vl> Is the vl specifier, encoded in "vl":

vl	<vl>
0	VLx2
1	VLx4



## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = width * (VL DIV esize);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
integer count = 0;

for e = 0 to elements-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    if last then count = count + 1;
    operand1 = operand1 + 1;

result = EncodePredCount(esize, elements, count, invert, PL);
PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert);
P[d, PL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILELO (predicate pair)

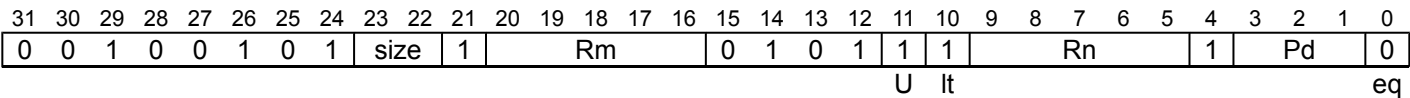
While incrementing unsigned scalar lower than scalar (pair of predicates)

Generate a pair of predicates that starting from the lowest numbered element of the pair is true while the incrementing value of the first, unsigned scalar operand is lower than the second scalar operand and false thereafter up to the highest numbered element of the pair.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2
(FEAT\_SVE2p1)



WHILELO { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d0 = UInt(Pd:'0');
integer d1 = UInt(Pd:'1');
boolean unsigned = TRUE;
SVEComp op = Comp_LT;
```

Assembler Symbols

<Pd1> Is the name of the first destination scalable predicate register, encoded as "Pd" times 2.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.

<Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL*2) mask = Ones(PL*2);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL*2) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = 0 to (elements*2)-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d0, PL] = result<PL-1:0>;
P[d1, PL] = result<PL*2-1:PL>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

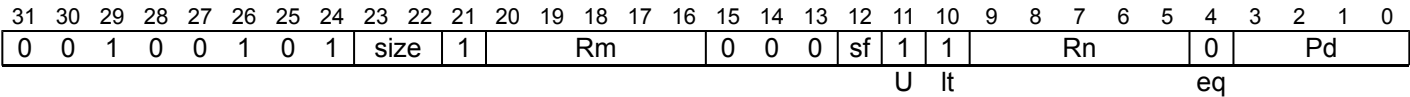
WHILELO (predicate)

While incrementing unsigned scalar lower than scalar

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, unsigned scalar operand is lower than the second scalar operand and false thereafter up to the highest numbered element.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



```
WHILELO <Pd>.<T>, <R><n>, <R><m>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = TRUE;
SVEComp op = Cmp_LT;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in "sf":

sf	<R>
0	W
1	X

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILELS (predicate as counter)

While incrementing unsigned scalar lower or same as scalar (predicate-as-counter)

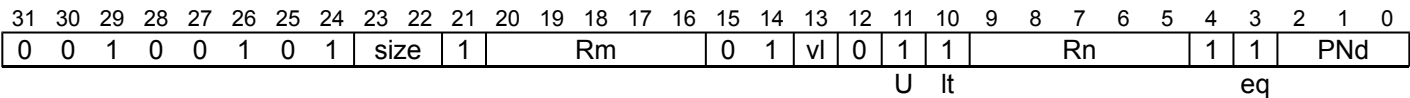
Generate a predicate for a group of two or four vectors that starting from the lowest numbered element of the group is true while the incrementing value of the first, unsigned scalar operand is lower or same as the second scalar operand and false thereafter up to the highest numbered element of the group.

If the second scalar operand is equal to the maximum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2
(FEAT\_SVE2p1)



WHILELS <PNd>.<T>, <Xn>, <Xm>, <vl>

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt('1':PNd);
boolean unsigned = TRUE;
boolean invert = FALSE;
SVEComp op = Cmp_LE;
integer width = 2 << UInt(vl);
```

Assembler Symbols

<PNd> Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

<vl> Is the vl specifier, encoded in "vl":

vl	<vl>
0	VLx2
1	VLx4

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = width * (VL DIV esize);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
integer count = 0;

for e = 0 to elements-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    if last then count = count + 1;
    operand1 = operand1 + 1;

result = EncodePredCount(esize, elements, count, invert, PL);
PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert);
P[d, PL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILELS (predicate pair)

While incrementing unsigned scalar lower or same as scalar (pair of predicates)

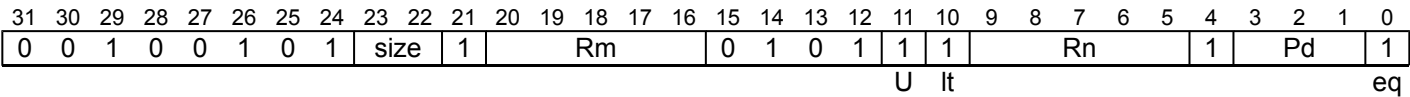
Generate a pair of predicates that starting from the lowest numbered element of the pair is true while the incrementing value of the first, unsigned scalar operand is lower or same as the second scalar operand and false thereafter up to the highest numbered element of the pair.

If the second scalar operand is equal to the maximum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2  
(FEAT\_SVE2p1)



```
WHILELS { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d0 = UInt(Pd:'0');
integer d1 = UInt(Pd:'1');
boolean unsigned = TRUE;
SVEComp op = Cmp_LE;
```

Assembler Symbols

<Pd1> Is the name of the first destination scalable predicate register, encoded as "Pd" times 2.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.

<Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL*2) mask = Ones(PL*2);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL*2) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = 0 to (elements*2)-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d0, PL] = result<PL-1:0>;
P[d1, PL] = result<PL*2-1:PL>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILELS (predicate)

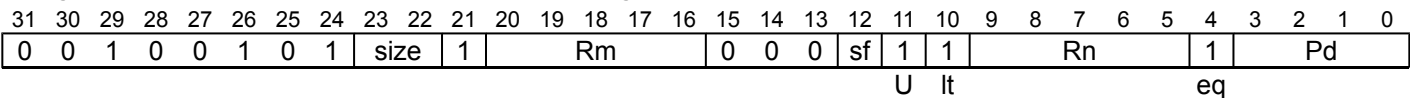
While incrementing unsigned scalar lower or same as scalar

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, unsigned scalar operand is lower or same as the second scalar operand and false thereafter up to the highest numbered element.

If the second scalar operand is equal to the maximum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



```
WHILELS <Pd>.<T>, <R><n>, <R><m>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = TRUE;
SVEComp op = Comp_LE;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in "sf":

sf	<R>
0	W
1	X

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILELT (predicate as counter)

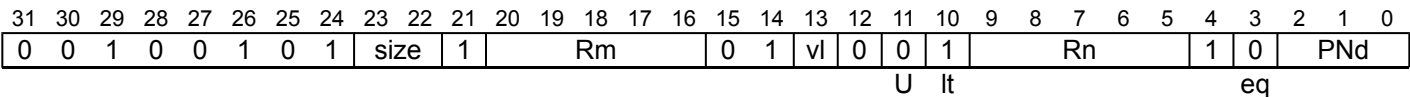
While incrementing signed scalar less than scalar (predicate-as-counter)

Generate a predicate for a group of two or four vectors that starting from the lowest numbered element of the group is true while the incrementing value of the first, signed scalar operand is less than the second scalar operand and false thereafter up to the highest numbered element of the group.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2
(FEAT\_SVE2p1)



WHILELT <PNd>.<T>, <Xn>, <Xm>, <vl>

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt('1':PNd);
boolean unsigned = FALSE;
boolean invert = FALSE;
SVEComp op = Cmp_LT;
integer width = 2 << UInt(vl);
```

Assembler Symbols

<PNd> Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

<vl> Is the vl specifier, encoded in "vl":

vl	<vl>
0	VLx2
1	VLx4

## Operation

```
if HaveSVE2p1() then CheckSVEEnabled(); else CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = width * (VL DIV esize);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
integer count = 0;

for e = 0 to elements-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    if last then count = count + 1;
    operand1 = operand1 + 1;

result = EncodePredCount(esize, elements, count, invert, PL);
PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert);
P[d, PL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILELT (predicate pair)

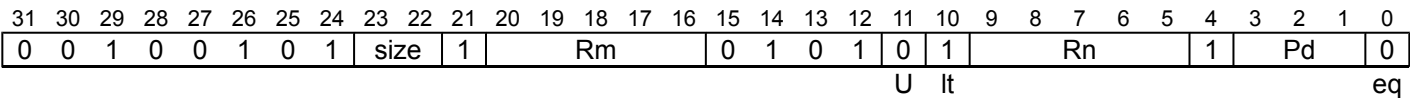
While incrementing signed scalar less than scalar (pair of predicates)

Generate a pair of predicates that starting from the lowest numbered element of the pair is true while the incrementing value of the first, signed scalar operand is less than the second scalar operand and false thereafter up to the highest numbered element of the pair.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2  
(FEAT\_SVE2p1)



```
WHILELT { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

```
if !HaveSME2() && !HaveSVE2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d0 = UInt(Pd:'0');
integer d1 = UInt(Pd:'1');
boolean unsigned = FALSE;
SVEComp op = Cmp_LT;
```

Assembler Symbols

- <Pd1> Is the name of the first destination scalable predicate register, encoded as "Pd" times 2.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D
- <Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.
- <Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL*2) mask = Ones(PL*2);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL*2) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = 0 to (elements*2)-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d0, PL] = result<PL-1:0>;
P[d1, PL] = result<PL*2-1:PL>;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

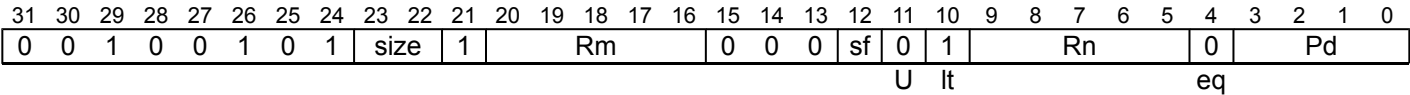
Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILELT (predicate)

While incrementing signed scalar less than scalar

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, signed scalar operand is less than the second scalar operand and false thereafter up to the highest numbered element. The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated. The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



```
WHILELT <Pd>.<T>, <R><n>, <R><m>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = FALSE;
SVEComp op = Cmp_LT;
```

Assembler Symbols

- <Pd>

Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D
- <R>

Is a width specifier, encoded in "sf":

sf	<R>
0	W
1	X
- <n>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.
- <m>

Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.



## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILERW

While free of read-after-write conflicts

This instruction checks two addresses for a conflict or overlap between address ranges of the form [addr,addr+VL÷8), where VL is the accessible vector length in bits, that could result in a loop-carried dependency through memory due to the use of these addresses by contiguous load and store instructions within the same iteration of a loop. Generate a predicate whose elements are true while the addresses cannot conflict within the same iteration, and false thereafter. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	0	1	0	1	size	1	Rm						0	0	1	1	0	0	Rn						1	Pd			
																															rw	

WHILERW <Pd>.<T>, <Xn>, <Xm>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D
- <Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(64) src1 = X[n, 64];
bits(64) src2 = X[m, 64];
integer operand1 = UInt(src1);
integer operand2 = UInt(src2);
bits(PL) result;
constant integer psize = esize DIV 8;

integer diff = Abs(operand2 - operand1) DIV (esize DIV 8);
for e = 0 to elements-1
    if diff == 0 || e < diff then
        Elem[result, e, psize] = ZeroExtend('1', psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WHILEWR

While free of write-after-read/write conflicts

This instruction checks two addresses for a conflict or overlap between address ranges of the form [addr,addr+VL÷8), where VL is the accessible vector length in bits, that could result in a loop-carried dependency through memory due to the use of these addresses by contiguous load and store instructions within the same iteration of a loop. Generate a predicate whose elements are true while the addresses cannot conflict within the same iteration, and false thereafter. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	0	1	0	1	size	1	Rm						0	0	1	1	0	0	Rn						0	Pd			
																														rw		

WHILEWR <Pd>.<T>, <Xn>, <Xm>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
```

Assembler Symbols

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D
- <Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(64) src1 = X[n, 64];
bits(64) src2 = X[m, 64];
integer operand1 = UInt(src1);
integer operand2 = UInt(src2);
bits(PL) result;
constant integer psize = esize DIV 8;

integer diff = (operand2 - operand1) DIV (esize DIV 8);
for e = 0 to elements-1
    if diff <= 0 || e < diff then
        Elem[result, e, psize] = ZeroExtend('1', psize);
    else
        Elem[result, e, psize] = ZeroExtend('0', psize);

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# WRFFR

Write the first-fault register

Read the source predicate register and place in the first-fault register (FFR). This instruction is intended to restore a saved FFR and is not recommended for general use by applications.

This instruction requires that the source predicate contains a MONOTONIC predicate value, in which starting from bit 0 there are zero or more 1 bits, followed only by 0 bits in any remaining bit positions. If the source is not a monotonic predicate value, then the resulting value in the FFR will be UNPREDICTABLE. It is not possible to generate a non-monotonic value in FFR when using SETFFR followed by first-fault or non-fault loads.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	0	1	0	1	0	0	0	1	0	0	1	0	0	0	Pn			0	0	0	0	0	

WRFFR <Pn>.B

```
if !HaveSVE() then UNDEFINED;
integer n = UInt(Pn);
```

## Assembler Symbols

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

## Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) operand = P[n, PL];

constant integer hsb = HighestSetBit(operand);
if hsb < 0 || IsOnes(operand<hsb:0>) then
    FFR[PL] = operand;
else // not a monotonic predicate
    FFR[PL] = bits(PL) UNKNOWN;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

XAR

Bitwise exclusive OR and rotate right by immediate

Bitwise exclusive OR the corresponding elements of the first and second source vectors, then rotate each result element right by an immediate amount. The final results are destructively placed in the corresponding elements of the destination and first source vector. This instruction is unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	tszh	1	tszl	imm3	0	0	1	1	0	1	Zm						Zdn							

```
XAR <Zdn>.<T>, <Zdn>.<T>, <Zm>.<T>, #<const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
if tsize == '0000' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
integer rot = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tszh:tszl:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = ROR(element1 EOR element2, rot);
Z[dn, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



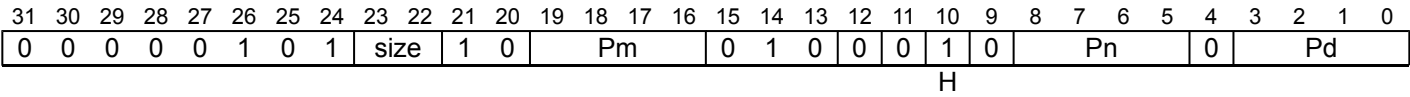
ZIP1, ZIP2 (predicates)

Interleave elements from two half predicates

Interleave alternating elements from the lowest or highest halves of the first and second source predicates and place in elements of the destination predicate. This instruction is unpredicated.

It has encodings from 2 classes: [High halves](#) and [Low halves](#)

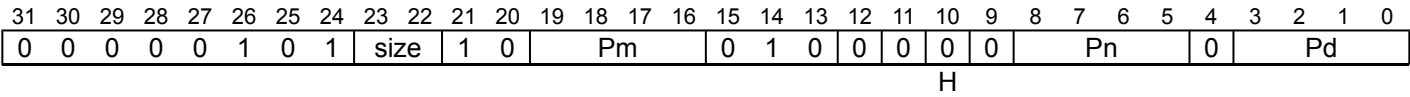
High halves



ZIP2 <Pd>.<T>, <Pn>.<T>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 1;
```

Low halves



ZIP1 <Pd>.<T>, <Pn>.<T>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 0;
```

Assembler Symbols

- <Pd>

Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D
- <Pn>

Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- <Pm>

Is the name of the second source scalable predicate register, encoded in the "Pm" field.

## Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (esize * 2);
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

integer base = part * pairs;
for p = 0 to pairs-1
    Elem[result, 2*p+0, esize DIV 8] = Elem[operand1, base+p, esize DIV 8];
    Elem[result, 2*p+1, esize DIV 8] = Elem[operand2, base+p, esize DIV 8];

P[d, PL] = result;
```

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ZIP1, ZIP2 (vectors)

Interleave elements from two half vectors

Interleave alternating elements from the lowest or highest halves of the first and second source vectors and place in elements of the destination vector. This instruction is unpredicated.  
The 128-bit element variant requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero. ID\_AA64ZFR0\_EL1.F64MM indicates whether the 128-bit element variant is implemented. The 128-bit element variant is illegal when executed in Streaming SVE mode, unless FEAT\_SME\_FA64 is implemented and enabled.  
It has encodings from 4 classes: [High halves](#) , [High halves \(quadwords\)](#) , [Low halves](#) and [Low halves \(quadwords\)](#)

High halves

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1				Zm			0	1	1	0	0	1				Zn				Zd		
H																															

ZIP2 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```

High halves (quadwords)  
(FEAT\_F64MM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	1	0	1			Zm			0	0	0	0	0	1				Zn				Zd		
H																															

ZIP2 <Zd>.Q, <Zn>.Q, <Zm>.Q

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```

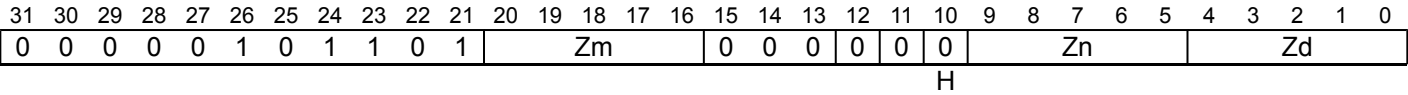
Low halves

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1				Zm			0	1	1	0	0	0				Zn				Zd		
H																															

ZIP1 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

Low halves (quadwords)  
(FEAT\_F64MM)



```
ZIP1 <Zd>.Q, <Zn>.Q, <Zm>.Q
```

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
if VL < esize * 2 then UNDEFINED;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Zeros(VL);

integer base = part * pairs;
for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

Z[d, VL] = result;
```

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

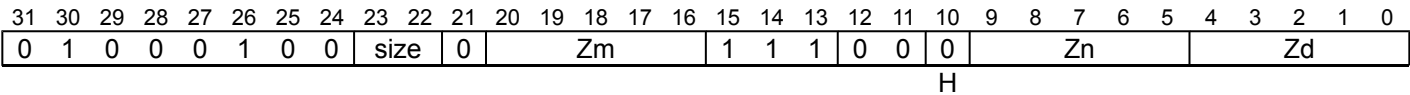
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

ZIPQ1

Interleave elements from low halves of each pair of quadword vector segments

Interleave alternating elements from low halves of the corresponding 128-bit vector segments of the first and second source vectors and place in elements of the corresponding destination vector segment. This instruction is unpredicated.

SVE2  
(FEAT\_SVE2p1)



```
ZIPQ1 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elements = 128 DIV esize;
constant integer pairs = elements DIV 2;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for s = 0 to segments-1
    integer base = s * elements + part * pairs;
    for p = 0 to pairs-1
        Elem[result, s * elements + 2 * p + 0, esize] = Elem[operand1, base + p, esize];
        Elem[result, s * elements + 2 * p + 1, esize] = Elem[operand2, base + p, esize];
Z[d, VL] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

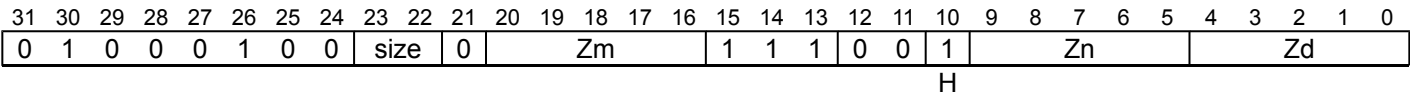
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ZIPQ2

Interleave elements from high halves of each pair of quadword vector segments

Interleave alternating elements from high halves of the corresponding 128-bit vector segments of the first and second source vectors and place in elements of the corresponding destination vector segment. This instruction is unpredicated.

SVE2  
(FEAT\_SVE2p1)



```
ZIPQ2 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE2p1() && !HaveSME2p1() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
constant integer elements = 128 DIV esize;
constant integer pairs = elements DIV 2;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for s = 0 to segments-1
    integer base = s * elements + part * pairs;
    for p = 0 to pairs-1
        Elem[result, s * elements + 2 * p + 0, esize] = Elem[operand1, base + p, esize];
        Elem[result, s * elements + 2 * p + 1, esize] = Elem[operand2, base + p, esize];

Z[d, VL] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## A64 -- SME Instructions (alphabetic order)

[ADD \(array accumulators\)](#): Add multi-vector to ZA array vector accumulators.

[ADD \(array results, multiple and single vector\)](#): Add replicated single vector to multi-vector with ZA array vector results.

[ADD \(array results, multiple vectors\)](#): Add multi-vector to multi-vector with ZA array vector results.

[ADD \(to vector\)](#): Add replicated single vector to multi-vector with multi-vector result.

[ADDHA](#): Add horizontally vector elements to ZA tile.

[ADDSPIL](#): Add multiple of Streaming SVE predicate register size to scalar register.

[ADDSPVL](#): Add multiple of Streaming SVE vector register size to scalar register.

[ADDVA](#): Add vertically vector elements to ZA tile.

[BFADD](#): BFloat16 floating-point add multi-vector to ZA array vector accumulators.

[BFCLAMP](#): Multi-vector BFloat16 floating-point clamp to minimum/maximum number.

[BFCVT](#): Multi-vector floating-point convert from single-precision to packed BFloat16 format.

[BFCVTN](#): Multi-vector floating-point convert from single-precision to interleaved BFloat16 format.

[BFDOT \(multiple and indexed vector\)](#): Multi-vector BFloat16 floating-point dot-product by indexed element.

[BFDOT \(multiple and single vector\)](#): Multi-vector BFloat16 floating-point dot-product by vector.

[BFDOT \(multiple vectors\)](#): Multi-vector BFloat16 floating-point dot-product.

[BFMAX \(multiple and single vector\)](#): Multi-vector BFloat16 floating-point maximum by vector.

[BFMAX \(multiple vectors\)](#): Multi-vector BFloat16 floating-point maximum.

[BFMAXNM \(multiple and single vector\)](#): Multi-vector BFloat16 floating-point maximum number by vector.

[BFMAXNM \(multiple vectors\)](#): Multi-vector BFloat16 floating-point maximum number.

[BFMIN \(multiple and single vector\)](#): Multi-vector BFloat16 floating-point minimum by vector.

[BFMIN \(multiple vectors\)](#): Multi-vector BFloat16 floating-point minimum.

[BFMINNM \(multiple and single vector\)](#): Multi-vector BFloat16 floating-point minimum number by vector.

[BFMINNM \(multiple vectors\)](#): Multi-vector BFloat16 floating-point minimum number.

[BFMLA \(multiple and indexed vector\)](#): Multi-vector BFloat16 floating-point fused multiply-add by indexed element.

[BFMLA \(multiple and single vector\)](#): Multi-vector BFloat16 floating-point fused multiply-add by vector.

[BFMLA \(multiple vectors\)](#): Multi-vector BFloat16 floating-point fused multiply-add.

[BFMLAL \(multiple and indexed vector\)](#): Multi-vector BFloat16 floating-point multiply-add long by indexed element.

[BFMLAL \(multiple and single vector\)](#): Multi-vector BFloat16 floating-point multiply-add long by vector.

[BFMLAL \(multiple vectors\)](#): Multi-vector BFloat16 floating-point multiply-add long.

[BFMLS \(multiple and indexed vector\)](#): Multi-vector BFloat16 floating-point fused multiply-subtract by indexed element.

[BFMLS \(multiple and single vector\)](#): Multi-vector BFloat16 floating-point fused multiply-subtract by vector.

[BFMLS \(multiple vectors\)](#): Multi-vector BFloat16 floating-point fused multiply-subtract.

[BFMLSL \(multiple and indexed vector\)](#): Multi-vector BFloat16 floating-point multiply-subtract long by indexed element.

[BFMLSL \(multiple and single vector\)](#): Multi-vector BFloat16 floating-point multiply-subtract long by vector.

[BFMLSL \(multiple vectors\)](#): Multi-vector BFloat16 floating-point multiply-subtract long.

[BFMOPA \(non-widening\)](#): BFloat16 floating-point outer product and accumulate.

[BFMOPA \(widening\)](#): BFloat16 sum of outer products and accumulate.

[BFMOPS \(non-widening\)](#): BFloat16 floating-point outer product and subtract.

[BFMOPS \(widening\)](#): BFloat16 sum of outer products and subtract.

[BFSUB](#): BFloat16 floating-point subtract multi-vector from ZA array vector accumulators.

[BFVDOT](#): Multi-vector BFloat16 floating-point vertical dot-product by indexed element.

[BMOPA](#): Bitwise exclusive NOR population count outer product and accumulate.

[BMOPS](#): Bitwise exclusive NOR population count outer product and subtract.

[FADD](#): Floating-point add multi-vector to ZA array vector accumulators.

[FCLAMP](#): Multi-vector floating-point clamp to minimum/maximum number.

[FCVT \(narrowing\)](#): Multi-vector floating-point convert from single-precision to packed half-precision.

[FCVT \(widening\)](#): Multi-vector floating-point convert from half-precision to single-precision (in-order).

[FCVTL](#): Multi-vector floating-point convert from half-precision to deinterleaved single-precision.

[FCVTN](#): Multi-vector floating-point convert from single-precision to interleaved half-precision.

[FCVTZS](#): Multi-vector floating-point convert to signed integer, rounding toward zero.

[FCVTZU](#): Multi-vector floating-point convert to unsigned integer, rounding toward zero.

[FDOT \(multiple and indexed vector\)](#): Multi-vector half-precision floating-point dot-product by indexed element.

[FDOT \(multiple and single vector\)](#): Multi-vector half-precision floating-point dot-product by vector.

[FDOT \(multiple vectors\)](#): Multi-vector half-precision floating-point dot-product.

[FMAX \(multiple and single vector\)](#): Multi-vector floating-point maximum by vector.

[FMAX \(multiple vectors\)](#): Multi-vector floating-point maximum.

[FMAXNM \(multiple and single vector\)](#): Multi-vector floating-point maximum number by vector.

[FMAXNM \(multiple vectors\)](#): Multi-vector floating-point maximum number.

[FMIN \(multiple and single vector\)](#): Multi-vector floating-point minimum by vector.

[FMIN \(multiple vectors\)](#): Multi-vector floating-point minimum.

[FMINNM \(multiple and single vector\)](#): Multi-vector floating-point minimum number by vector.

[FMINNM \(multiple vectors\)](#): Multi-vector floating-point minimum number.

[FMLA \(multiple and indexed vector\)](#): Multi-vector floating-point fused multiply-add by indexed element.

[FMLA \(multiple and single vector\)](#): Multi-vector floating-point fused multiply-add by vector.

[FMLA \(multiple vectors\)](#): Multi-vector floating-point fused multiply-add.

[FMLAL \(multiple and indexed vector\)](#): Multi-vector floating-point multiply-add long by indexed element.

[FMLAL \(multiple and single vector\)](#): Multi-vector floating-point multiply-add long by vector.

[FMLAL \(multiple vectors\)](#): Multi-vector floating-point multiply-add long.

[FMLS \(multiple and indexed vector\)](#): Multi-vector floating-point fused multiply-subtract by indexed element.

[FMLS \(multiple and single vector\)](#): Multi-vector floating-point fused multiply-subtract by vector.

[FMLS \(multiple vectors\)](#): Multi-vector floating-point fused multiply-subtract.

[FMLS \(multiple and indexed vector\)](#): Multi-vector floating-point multiply-subtract long by indexed element.

[FMLS \(multiple and single vector\)](#): Multi-vector floating-point multiply-subtract long by vector.

[FMLS \(multiple vectors\)](#): Multi-vector floating-point multiply-subtract long.

[FMOPA \(non-widening\)](#): Floating-point outer product and accumulate.

[FMOPA \(widening\)](#): Half-precision floating-point sum of outer products and accumulate.

[FMOPS \(non-widening\)](#): Floating-point outer product and subtract.

[FMOPS \(widening\)](#): Half-precision floating-point sum of outer products and subtract.

[FRINTA](#): Multi-vector floating-point round to integral value, to nearest with ties away from zero.

[FRINTM](#): Multi-vector floating-point round to integral value, toward minus Infinity.

[FRINTN](#): Multi-vector floating-point round to integral value, to nearest with ties to even.

[FRINTP](#): Multi-vector floating-point round to integral value, toward plus Infinity.

[FSUB](#): Floating-point subtract multi-vector from ZA array vector accumulators.

[FVDOT](#): Multi-vector half-precision floating-point vertical dot-product by indexed element.

[LD1B \(scalar plus immediate, strided registers\)](#): Contiguous load of bytes to multiple strided vectors (immediate index).

[LD1B \(scalar plus scalar, strided registers\)](#): Contiguous load of bytes to multiple strided vectors (scalar index).

[LD1B \(scalar plus scalar, tile slice\)](#): Contiguous load of bytes to 8-bit element ZA tile slice.

[LD1D \(scalar plus immediate, strided registers\)](#): Contiguous load of doublewords to multiple strided vectors (immediate index).

[LD1D \(scalar plus scalar, strided registers\)](#): Contiguous load of doublewords to multiple strided vectors (scalar index).

[LD1D \(scalar plus scalar, tile slice\)](#): Contiguous load of doublewords to 64-bit element ZA tile slice.

[LD1H \(scalar plus immediate, strided registers\)](#): Contiguous load of halfwords to multiple strided vectors (immediate index).

[LD1H \(scalar plus scalar, strided registers\)](#): Contiguous load of halfwords to multiple strided vectors (scalar index).

[LD1H \(scalar plus scalar, tile slice\)](#): Contiguous load of halfwords to 16-bit element ZA tile slice.

[LD1Q](#): Contiguous load of quadwords to 128-bit element ZA tile slice.

[LD1W \(scalar plus immediate, strided registers\)](#): Contiguous load of words to multiple strided vectors (immediate index).

[LD1W \(scalar plus scalar, strided registers\)](#): Contiguous load of words to multiple strided vectors (scalar index).

[LD1W \(scalar plus scalar, tile slice\)](#): Contiguous load of words to 32-bit element ZA tile slice.

[LDNT1B \(scalar plus immediate, strided registers\)](#): Contiguous load non-temporal of bytes to multiple strided vectors (immediate index).

[LDNT1B \(scalar plus scalar, strided registers\)](#): Contiguous load non-temporal of bytes to multiple strided vectors (scalar index).

[LDNT1D \(scalar plus immediate, strided registers\)](#): Contiguous load non-temporal of doublewords to multiple strided vectors (immediate index).

[LDNT1D \(scalar plus scalar, strided registers\)](#): Contiguous load non-temporal of doublewords to multiple strided vectors (scalar index).

[LDNT1H \(scalar plus immediate, strided registers\)](#): Contiguous load non-temporal of halfwords to multiple strided vectors (immediate index).

[LDNT1H \(scalar plus scalar, strided registers\)](#): Contiguous load non-temporal of halfwords to multiple strided vectors (scalar index).

[LDNT1W \(scalar plus immediate, strided registers\)](#): Contiguous load non-temporal of words to multiple strided vectors (immediate index).

[LDNT1W \(scalar plus scalar, strided registers\)](#): Contiguous load non-temporal of words to multiple strided vectors (scalar index).

[LDR \(vector\)](#): Load ZA array vector.

[LDR \(ZT0\)](#): Load ZT0 register.

[LUTI2 \(four registers\)](#): Lookup table read with 2-bit indexes.

[LUTI2 \(single\)](#): Lookup table read with 2-bit indexes.

[LUTI2 \(two registers\)](#): Lookup table read with 2-bit indexes.

[LUTI4 \(four registers\)](#): Lookup table read with 4-bit indexes.

[LUTI4 \(single\)](#): Lookup table read with 4-bit indexes.

[LUTI4 \(two registers\)](#): Lookup table read with 4-bit indexes.

[MOV \(array to vector, four registers\)](#): Move four ZA single-vector groups to four vector registers: an alias of MOVA (array to vector, four registers).

[MOV \(array to vector, two registers\)](#): Move two ZA single-vector groups to two vector registers: an alias of MOVA (array to vector, two registers).

[MOV \(tile to vector, four registers\)](#): Move four ZA tile slices to four vector registers: an alias of MOVA (tile to vector, four registers).

[MOV \(tile to vector, single\)](#): Move ZA tile slice to vector register: an alias of MOVA (tile to vector, single).

[MOV \(tile to vector, two registers\)](#): Move two ZA tile slices to two vector registers: an alias of MOVA (tile to vector, two registers).

[MOV \(vector to array, four registers\)](#): Move four vector registers to four ZA single-vector groups: an alias of MOVA (vector to array, four registers).

[MOV \(vector to array, two registers\)](#): Move two vector registers to two ZA single-vector groups: an alias of MOVA (vector to array, two registers).

[MOV \(vector to tile, four registers\)](#): Move four vector registers to four ZA tile slices: an alias of MOVA (vector to tile, four registers).

[MOV \(vector to tile, single\)](#): Move vector register to ZA tile slice: an alias of MOVA (vector to tile, single).

[MOV \(vector to tile, two registers\)](#): Move two vector registers to two ZA tile slices: an alias of MOVA (vector to tile, two registers).

[MOVA \(array to vector, four registers\)](#): Move four ZA single-vector groups to four vector registers.

[MOVA \(array to vector, two registers\)](#): Move two ZA single-vector groups to two vector registers.

[MOVA \(tile to vector, four registers\)](#): Move four ZA tile slices to four vector registers.

[MOVA \(tile to vector, single\)](#): Move ZA tile slice to vector register.

[MOVA \(tile to vector, two registers\)](#): Move two ZA tile slices to two vector registers.

[MOVA \(vector to array, four registers\)](#): Move four vector registers to four ZA single-vector groups.

[MOVA \(vector to array, two registers\)](#): Move two vector registers to two ZA single-vector groups.

[MOVA \(vector to tile, four registers\)](#): Move four vector registers to four ZA tile slices.

[MOVA \(vector to tile, single\)](#): Move vector register to ZA tile slice.

[MOVA \(vector to tile, two registers\)](#): Move two vector registers to two ZA tile slices.

[MOVAZ \(array to vector, four registers\)](#): Move and zero four ZA single-vector groups to vector registers.

[MOVAZ \(array to vector, two registers\)](#): Move and zero two ZA single-vector groups to vector registers.

[MOVAZ \(tile to vector, four registers\)](#): Move and zero four ZA tile slices to vector registers.

[MOVAZ \(tile to vector, single\)](#): Move and zero ZA tile slice to vector register.

[MOVAZ \(tile to vector, two registers\)](#): Move and zero two ZA tile slices to vector registers.

[MOVT \(scalar to ZT0\)](#): Move 8 bytes from general-purpose register to ZT0.

[MOVT \(ZT0 to scalar\)](#): Move 8 bytes from ZT0 to general-purpose register.

[RDSVL](#): Read multiple of Streaming SVE vector register size to scalar register.

[SCLAMP](#): Multi-vector signed clamp to minimum/maximum vector.

[SCVTF](#): Multi-vector signed integer convert to floating-point.

[SDOT \(2-way, multiple and indexed vector\)](#): Multi-vector signed integer dot-product by indexed element.

[SDOT \(2-way, multiple and single vector\)](#): Multi-vector signed integer dot-product by vector.

[SDOT \(2-way, multiple vectors\)](#): Multi-vector signed integer dot-product.

[SDOT \(4-way, multiple and indexed vector\)](#): Multi-vector signed integer dot-product by indexed element.

[SDOT \(4-way, multiple and single vector\)](#): Multi-vector signed integer dot-product by vector.

[SDOT \(4-way, multiple vectors\)](#): Multi-vector signed integer dot-product.

[SEL](#): Multi-vector conditionally select elements from two vectors.

[SMAX \(multiple and single vector\)](#): Multi-vector signed maximum by vector.

[SMAX \(multiple vectors\)](#): Multi-vector signed maximum.

[SMIN \(multiple and single vector\)](#): Multi-vector signed minimum by vector.

[SMIN \(multiple vectors\)](#): Multi-vector signed minimum.

[SMLAL \(multiple and indexed vector\)](#): Multi-vector signed integer multiply-add long by indexed element.

[SMLAL \(multiple and single vector\)](#): Multi-vector signed integer multiply-add long by vector.

[SMLAL \(multiple vectors\)](#): Multi-vector signed integer multiply-add long.

[SMLALL \(multiple and indexed vector\)](#): Multi-vector signed integer multiply-add long-long by indexed element.

[SMLALL \(multiple and single vector\)](#): Multi-vector signed integer multiply-add long-long by vector.

[SMLALL \(multiple vectors\)](#): Multi-vector signed integer multiply-add long-long.

[SMLSL \(multiple and indexed vector\)](#): Multi-vector signed integer multiply-subtract long by indexed element.

[SMLSL \(multiple and single vector\)](#): Multi-vector signed integer multiply-subtract long by vector.

[SMLSL \(multiple vectors\)](#): Multi-vector signed integer multiply-subtract long.

[SMLSLL \(multiple and indexed vector\)](#): Multi-vector signed integer multiply-subtract long-long by indexed element.

[SMLSLL \(multiple and single vector\)](#): Multi-vector signed integer multiply-subtract long-long by vector.

[SMLSLL \(multiple vectors\)](#): Multi-vector signed integer multiply-subtract long-long.

[SMOPA \(2-way\)](#): Signed integer sum of outer products and accumulate.

[SMOPA \(4-way\)](#): Signed integer sum of outer products and accumulate.

[SMOPS \(2-way\)](#): Signed integer sum of outer products and subtract.

[SMOPS \(4-way\)](#): Signed integer sum of outer products and subtract.

[SQCVT \(four registers\)](#): Multi-vector signed saturating extract narrow.

[SQCVT \(two registers\)](#): Multi-vector signed saturating extract narrow.

[SQCVTN](#): Multi-vector signed saturating extract narrow and interleave.

[SQCVTU \(four registers\)](#): Multi-vector signed saturating unsigned extract narrow.

[SQCVTU \(two registers\)](#): Multi-vector signed saturating unsigned extract narrow.

[SQCVTUN](#): Multi-vector signed saturating unsigned extract narrow and interleave.

[SQDMULH \(multiple and single vector\)](#): Multi-vector signed saturating doubling multiply high by vector.

[SQDMULH \(multiple vectors\)](#): Multi-vector signed saturating doubling multiply high.

[SQRSHR \(four registers\)](#): Multi-vector signed saturating rounding shift right narrow by immediate.

[SQRSHR \(two registers\)](#): Multi-vector signed saturating rounding shift right narrow by immediate.

[SQRSHRN](#): Multi-vector signed saturating rounding shift right narrow by immediate and interleave.

[SQRSHRU \(four registers\)](#): Multi-vector signed saturating rounding shift right unsigned narrow by immediate.

[SQRSHRU \(two registers\)](#): Multi-vector signed saturating rounding shift right unsigned narrow by immediate.

[SQRSHRUN](#): Multi-vector signed saturating rounding shift right unsigned narrow by immediate and interleave.

[SRSHL \(multiple and single vector\)](#): Multi-vector signed rounding shift left by vector.

[SRSHL \(multiple vectors\)](#): Multi-vector signed rounding shift left.

[ST1B \(scalar plus immediate, strided registers\)](#): Contiguous store of bytes from multiple strided vectors (immediate index).

[ST1B \(scalar plus scalar, strided registers\)](#): Contiguous store of bytes from multiple strided vectors (scalar index).

[ST1B \(scalar plus scalar, tile slice\)](#): Contiguous store of bytes from 8-bit element ZA tile slice.

[ST1D \(scalar plus immediate, strided registers\)](#): Contiguous store of doublewords from multiple strided vectors (immediate index).

[ST1D \(scalar plus scalar, strided registers\)](#): Contiguous store of doublewords from multiple strided vectors (scalar index).

[ST1D \(scalar plus scalar, tile slice\)](#): Contiguous store of doublewords from 64-bit element ZA tile slice.

[ST1H \(scalar plus immediate, strided registers\)](#): Contiguous store of halfwords from multiple strided vectors (immediate index).

[ST1H \(scalar plus scalar, strided registers\)](#): Contiguous store of halfwords from multiple strided vectors (scalar index).

[ST1H \(scalar plus scalar, tile slice\)](#): Contiguous store of halfwords from 16-bit element ZA tile slice.

[ST1Q](#): Contiguous store of quadwords from 128-bit element ZA tile slice.

[ST1W \(scalar plus immediate, strided registers\)](#): Contiguous store of words from multiple strided vectors (immediate index).

[ST1W \(scalar plus scalar, strided registers\)](#): Contiguous store of words from multiple strided vectors (scalar index).

[ST1W \(scalar plus scalar, tile slice\)](#): Contiguous store of words from 32-bit element ZA tile slice.



[STNT1B \(scalar plus immediate, strided registers\)](#): Contiguous store non-temporal of bytes from multiple strided vectors (immediate index).

[STNT1B \(scalar plus scalar, strided registers\)](#): Contiguous store non-temporal of bytes from multiple strided vectors (scalar index).

[STNT1D \(scalar plus immediate, strided registers\)](#): Contiguous store non-temporal of doublewords from multiple strided vectors (immediate index).

[STNT1D \(scalar plus scalar, strided registers\)](#): Contiguous store non-temporal of doublewords from multiple strided vectors (scalar index).

[STNT1H \(scalar plus immediate, strided registers\)](#): Contiguous store non-temporal of halfwords from multiple strided vectors (immediate index).

[STNT1H \(scalar plus scalar, strided registers\)](#): Contiguous store non-temporal of halfwords from multiple strided vectors (scalar index).

[STNT1W \(scalar plus immediate, strided registers\)](#): Contiguous store non-temporal of words from multiple strided vectors (immediate index).

[STNT1W \(scalar plus scalar, strided registers\)](#): Contiguous store non-temporal of words from multiple strided vectors (scalar index).

[STR \(vector\)](#): Store ZA array vector.

[STR \(ZT0\)](#): Store ZT0 register.

[SUB \(array accumulators\)](#): Subtract multi-vector from ZA array vector accumulators.

[SUB \(array results, multiple and single vector\)](#): Subtract replicated single vector from multi-vector with ZA array vector results.

[SUB \(array results, multiple vectors\)](#): Subtract multi-vector from multi-vector with ZA array vector results.

[SUDOT \(multiple and indexed vector\)](#): Multi-vector signed by unsigned integer dot-product by indexed elements.

[SUDOT \(multiple and single vector\)](#): Multi-vector signed by unsigned integer dot-product by vector.

[SUMLALL \(multiple and indexed vector\)](#): Multi-vector signed by unsigned integer multiply-add long-long by indexed element.

[SUMLALL \(multiple and single vector\)](#): Multi-vector signed by unsigned integer multiply-add long-long by vector.

[SUMOPA](#): Signed by unsigned integer sum of outer products and accumulate.

[SUMOPS](#): Signed by unsigned integer sum of outer products and subtract.

[SUNPK](#): Unpack and sign-extend multi-vector elements.

[SUVDOT](#): Multi-vector signed by unsigned integer vertical dot-product by indexed element.

[SVDOT \(2-way\)](#): Multi-vector signed integer vertical dot-product by indexed element.

[SVDOT \(4-way\)](#): Multi-vector signed integer vertical dot-product by indexed element.

[UCLAMP](#): Multi-vector unsigned clamp to minimum/maximum vector.

[UCVTE](#): Multi-vector unsigned integer convert to floating-point.

[UDOT \(2-way, multiple and indexed vector\)](#): Multi-vector unsigned integer dot-product by indexed element.

[UDOT \(2-way, multiple and single vector\)](#): Multi-vector unsigned integer dot-product by vector.

[UDOT \(2-way, multiple vectors\)](#): Multi-vector unsigned integer dot-product.

[UDOT \(4-way, multiple and indexed vector\)](#): Multi-vector unsigned integer dot-product by indexed element.

[UDOT \(4-way, multiple and single vector\)](#): Multi-vector unsigned integer dot-product by vector.

[UDOT \(4-way, multiple vectors\)](#): Multi-vector unsigned integer dot-product.

[UMAX \(multiple and single vector\)](#): Multi-vector unsigned maximum by vector.

[UMAX \(multiple vectors\)](#): Multi-vector unsigned maximum.

[UMIN \(multiple and single vector\)](#): Multi-vector unsigned minimum by vector.

[UMIN \(multiple vectors\)](#): Multi-vector unsigned minimum.

[UMLAL \(multiple and indexed vector\)](#): Multi-vector unsigned integer multiply-add long by indexed element.

[UMLAL \(multiple and single vector\)](#): Multi-vector unsigned integer multiply-add long by vector.

[UMLAL \(multiple vectors\)](#): Multi-vector unsigned integer multiply-add long.

[UMLALL \(multiple and indexed vector\)](#): Multi-vector unsigned integer multiply-add long-long by indexed element.

[UMLALL \(multiple and single vector\)](#): Multi-vector unsigned integer multiply-add long-long by vector.

[UMLALL \(multiple vectors\)](#): Multi-vector unsigned integer multiply-add long-long.

[UMLSL \(multiple and indexed vector\)](#): Multi-vector unsigned integer multiply-subtract long by indexed element.

[UMLSL \(multiple and single vector\)](#): Multi-vector unsigned integer multiply-subtract long by vector.

[UMLSL \(multiple vectors\)](#): Multi-vector unsigned integer multiply-subtract long.

[UMLSLL \(multiple and indexed vector\)](#): Multi-vector unsigned integer multiply-subtract long-long by indexed element.

[UMLSLL \(multiple and single vector\)](#): Multi-vector unsigned integer multiply-subtract long-long by vector.

[UMLSLL \(multiple vectors\)](#): Multi-vector unsigned integer multiply-subtract long-long.

[UMOPA \(2-way\)](#): Unsigned integer sum of outer products and accumulate.

[UMOPA \(4-way\)](#): Unsigned integer sum of outer products and accumulate.

[UMOPS \(2-way\)](#): Unsigned integer sum of outer products and subtract.

[UMOPS \(4-way\)](#): Unsigned integer sum of outer products and subtract.

[UQCVT \(four registers\)](#): Multi-vector unsigned saturating extract narrow.

[UQCVT \(two registers\)](#): Multi-vector unsigned saturating extract narrow.

[UQCVTN](#): Multi-vector unsigned saturating extract narrow and interleave.

[UQRSHR \(four registers\)](#): Multi-vector unsigned saturating rounding shift right narrow by immediate.

[UQRSHR \(two registers\)](#): Multi-vector unsigned saturating rounding shift right narrow by immediate.

[UQRSHRN](#): Multi-vector unsigned saturating rounding shift right narrow by immediate and interleave.

[URSHL \(multiple and single vector\)](#): Multi-vector unsigned rounding shift left by vector.

[URSHL \(multiple vectors\)](#): Multi-vector unsigned rounding shift left.

[USDOT \(multiple and indexed vector\)](#): Multi-vector unsigned by signed integer dot-product by indexed element.

[USDOT \(multiple and single vector\)](#): Multi-vector unsigned by signed integer dot-product by vector.

[USDOT \(multiple vectors\)](#): Multi-vector unsigned by signed integer dot-product.

[USMLALL \(multiple and indexed vector\)](#): Multi-vector unsigned by signed integer multiply-add long-long by indexed element.

[USMLALL \(multiple and single vector\)](#): Multi-vector unsigned by signed integer multiply-add long-long by vector.

[USMLALL \(multiple vectors\)](#): Multi-vector unsigned by signed integer multiply-add long-long.

[USMOPA](#): Unsigned by signed integer sum of outer products and accumulate.

[USMOPS](#): Unsigned by signed integer sum of outer products and subtract.



[USVDOT](#): Multi-vector unsigned by signed integer vertical dot-product by indexed element.

[UUNPK](#): Unpack and zero-extend multi-vector elements.

[UVDOT \(2-way\)](#): Multi-vector unsigned integer vertical dot-product by indexed element.

[UVDOT \(4-way\)](#): Multi-vector unsigned integer vertical dot-product by indexed element.

[UZP \(four registers\)](#): Concatenate elements from four vectors.

[UZP \(two registers\)](#): Concatenate elements from two vectors.

[ZERO \(double-vector\)](#): Zero ZA double-vector groups.

[ZERO \(quad-vector\)](#): Zero ZA quad-vector groups.

[ZERO \(single-vector\)](#): Zero ZA single-vector groups.

[ZERO \(tile\)](#): Zero a list of 64-bit element ZA tiles.

[ZERO \(ZT0\)](#): Zero ZT0.

[ZIP \(four registers\)](#): Interleave elements from four vectors.

[ZIP \(two registers\)](#): Interleave elements from two vectors.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD (array accumulators)

Add multi-vector to ZA array vector accumulators

Destructively add all elements of the two or four source vectors to the corresponding elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

Two ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	0	0	Rv	1	1	1	Zm			0	1	0	off3				
																											S				

ADD ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	1	0	Rv	1	1	1	Zm			0	0	1	0	off3			
																											S				

ADD ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.
- For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = ZAvector[vec, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = element1 + element2;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD (array results, multiple and single vector)

Add replicated single vector to multi-vector with ZA array vector results

Add all corresponding elements of the second source vector and the two or four first source vectors and place the results in the corresponding elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

Two ZA single-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm			0	Rv	1	1	0	Zn			1	0	off3			S			

ADD ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm			0	Rv	1	1	0	Zn			1	0	off3			S			

ADD ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1>	Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = element1 + element2;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD (array results, multiple vectors)

Add multi-vector to multi-vector with ZA array vector results

Add all corresponding elements of the two or four second source vectors and first source vectors and place the results in the corresponding elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

Two ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1		Zm			0	0	Rv		1	1	0		Zn			0	1	0		off3	
																S															

ADD ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm		0		1	0	Rv		1		1	0	Zn		0		0	1	0	off3	
																S															

ADD ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1>	For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm1>	For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
<Zm4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
<Zm2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = element1 + element2;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD (to vector)

Add replicated single vector to multi-vector with multi-vector result

Add elements of the second source vector to the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0	Zm				1	0	1	0	0	0	1	1	0	0	0	Zdn				0	

ADD { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt('0':Zm);
constant integer nreg = 2;
```

Four registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0	Zm				1	0	1	0	1	0	1	1	0	0	0	Zdn			0	0	

ADD { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt('0':Zm);
constant integer nreg = 4;
```

Assembler Symbols

- <Zdn1>

For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <T>

Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D
- <Zdn4>

Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2>

Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.



## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[results[r], e, esize] = element1 + element2;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADDHA

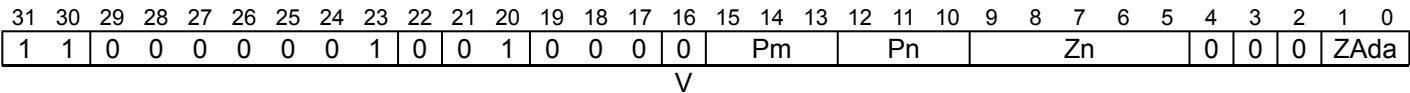
Add horizontally vector elements to ZA tile

Add each element of the source vector to the corresponding active element of each horizontal slice of a ZA tile. The tile elements are predicated by a pair of governing predicates. An element of a horizontal slice is considered active if its corresponding element in the second governing predicate is TRUE and the element corresponding to its horizontal slice number in the first governing predicate is TRUE. Inactive elements in the destination tile remain unmodified.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

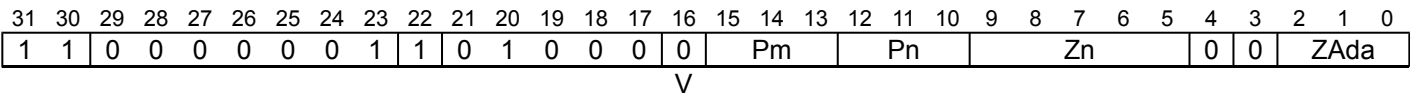
32-bit  
(FEAT\_SME)



ADDHA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer da = UInt(ZAda);
```

64-bit  
(FEAT\_SME\_I16I64)



ADDHA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer da = UInt(ZAda);
```

Assembler Symbols

- <ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.  
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand_src = Z[n, VL];
bits(dim*dim*esize) operand_acc = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for col = 0 to dim-1
    bits(esize) element = Elem[operand_src, col, esize];
    for row = 0 to dim-1
        bits(esize) res = Elem[operand_acc, row*dim+col, esize];
        if (ActivePredicateElement(mask1, row, esize) &&
            ActivePredicateElement(mask2, col, esize)) then
            res = res + element;
        Elem[result, row*dim+col, esize] = res;

ZAtile[da, esize, dim*dim*esize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADDSPL

Add multiple of Streaming SVE predicate register size to scalar register

Add the Streaming SVE predicate register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer and place the result in the 64-bit destination general-purpose register or current stack pointer.

This instruction does not require the PE to be in Streaming SVE mode.

SME  
(FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	1	1	Rn				0	1	0	1	1	imm6						Rd					

ADDSPL <Xd|SP>, <Xn|SP>, #<imm>

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm6);
```

Assembler Symbols

- <Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

```
CheckSMEEnabled();
constant integer SVL = CurrentSVL;
integer len = imm * (SVL DIV 64);
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) result = operand1 + len;

if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

ADDSVL

Add multiple of Streaming SVE vector register size to scalar register

Add the Streaming SVE vector register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer, and place the result in the 64-bit destination general-purpose register or current stack pointer.  
This instruction does not require the PE to be in Streaming SVE mode.

SME  
(FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	1	Rn					0	1	0	1	1	imm6						Rd				

ADDSVL <Xd|SP>, <Xn|SP>, #<imm>

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm6);
```

Assembler Symbols

- <Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- <Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- <imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

```
CheckSMEEnabled();
constant integer SVL = CurrentSVL;
integer len = imm * (SVL DIV 8);
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) result = operand1 + len;

if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

ADDVA

Add vertically vector elements to ZA tile

Add each element of the source vector to the corresponding active element of each vertical slice of a ZA tile. The tile elements are predicated by a pair of governing predicates. An element of a vertical slice is considered active if its corresponding element in the first governing predicate is TRUE and the element corresponding to its vertical slice number in the second governing predicate is TRUE. Inactive elements in the destination tile remain unmodified. ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 64-bit integer variant is implemented. It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

32-bit  
(FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	1	0	0	0	1	Pm			Pn			Zn			0			0	0	ZAda	
V																															

ADDVA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer da = UInt(ZAda);
```

64-bit  
(FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	1	0	0	0	1	Pm			Pn			Zn			0			0	ZAda		
V																															

ADDVA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer da = UInt(ZAda);
```

Assembler Symbols

- <ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.  
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand_src = Z[n, VL];
bits(dim*dim*esize) operand_acc = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for row = 0 to dim-1
    bits(esize) element = Elem[operand_src, row, esize];
    for col = 0 to dim-1
        bits(esize) res = Elem[operand_acc, row*dim+col, esize];
        if (ActivePredicateElement(mask1, row, esize) &&
            ActivePredicateElement(mask2, col, esize)) then
            res = res + element;
        Elem[result, row*dim+col, esize] = res;

ZAtile[da, esize, dim*dim*esize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BFADD

BFloat16 floating-point add multi-vector to ZA array vector accumulators

Destructively add all elements of the two or four source vectors to the corresponding BFloat16 elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME2.1 ZA-targeting non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

## Two ZA single-vectors (FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	0	0	1	0	0	0	Rv	1	1	1	Zm			0	0	0	off3				
SZ																S															

BFADD ZA.H[<Wv>, <offs>{, VGx2}], { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer v = UInt('010':Rv);
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

## Four ZA single-vectors (FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	1	1	0	0	1	0	1	0	Rv	1	1	1	Zm			0			0	0	0	off3		
SZ																S																

BFADD ZA.H[<Wv>, <offs>{, VGx4}], { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer v = UInt('010':Rv);
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

## Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  
For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.



## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
  bits(VL) operand1 = ZAvector[vec, VL];
  bits(VL) operand2 = Z[m+r, VL];
  for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, e, 16];
    bits(16) element2 = Elem[operand2, e, 16];
    Elem[result, e, 16] = BFAdd ZA(element1, element2, FPCR[]);
  ZAvector[vec, VL] = result;
  vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BFCLAMP

Multi-vector BFloat16 floating-point clamp to minimum/maximum number

Clamp each BFloat16 element in the two or four destination vectors to between the BFloat16 minimum value in the corresponding element of the first source vector and the BFloat16 maximum value in the corresponding element of the second source vector and destructively place the clamped results in the corresponding elements of the two or four destination vectors.

Regardless of the value of FPCR.AH, the behavior is as follows for each minimum number and maximum number operation:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction follows SME2.1 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

## Two registers

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	Zm				1	1	0	0	0	0	Zn				Zd				0		
size<1>size<0>																															

BFCLAMP { <Zd1>.H-<Zd2>.H }, <Zn>.H, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd:'0');
constant integer nreg = 2;
```

## Four registers

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	Zm				1	1	0	0	1	0	Zn				Zd				0	0	
size<1>size<0>																															

BFCLAMP { <Zd1>.H-<Zd4>.H }, <Zn>.H, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd:'00');
constant integer nreg = 4;
```

## Assembler Symbols

- <Zd1>

For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.  
  
For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <Zd4>

Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = Z[d+r, VL];
    for e = 0 to elements-1
        bits(16) element1 = Elem[operand1, e, 16];
        bits(16) element2 = Elem[operand2, e, 16];
        bits(16) element3 = Elem[operand3, e, 16];
        Elem[results[r], e, 16] = BFMinNum(BFMaxNum(element1, element3, FPCR[]), element2, FPCR[]);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

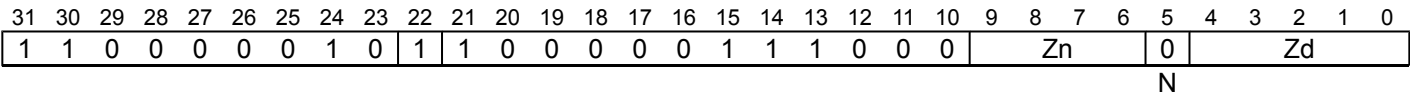
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFCVT

Multi-vector floating-point convert from single-precision to packed BFloat16 format

Convert to BFloat16 from single-precision, each element of the two source vectors, and place the results in the half-width destination elements.  
This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.  
This instruction is unpredicated.

SME2  
(FEAT\_SME2)



```
BFCVT <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
bits(VL) result;

bits(VL) operand1 = Z[n+0, VL];
bits(VL) operand2 = Z[n+1, VL];
for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, e, 32];
    bits(32) element2 = Elem[operand2, e, 32];
    bits(16) res1 = FPConvertBF(element1, FPCR[]);
    bits(16) res2 = FPConvertBF(element2, FPCR[]);
    Elem[result, e, 16] = res1;
    Elem[result, elements+e, 16] = res2;

Z[d, VL] = result;
```

BFCVTN

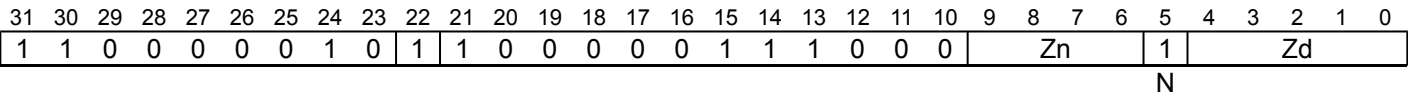
Multi-vector floating-point convert from single-precision to interleaved BFloat16 format

Convert to BFloat16 from single-precision, each element of the two source vectors, and place the two-way interleaved results in the half-width destination elements.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

SME2
(FEAT\_SME2)



BFCVTN <Zd>.H, { <Zn1>.S-<Zn2>.S }

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
bits(VL) result;

bits(VL) operand1 = Z[n+0, VL];
bits(VL) operand2 = Z[n+1, VL];
for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, e, 32];
    bits(32) element2 = Elem[operand2, e, 32];
    bits(16) res1 = FPConvertBF(element1, FPCR[]);
    bits(16) res2 = FPConvertBF(element2, FPCR[]);
    Elem[result, 2*e + 0, 16] = res1;
    Elem[result, 2*e + 1, 16] = res2;

Z[d, VL] = result;
```

# BFDOT (multiple and indexed vector)

Multi-vector BFloat16 floating-point dot-product by indexed element

The instruction computes the dot product of a pair of BF16 values held in the corresponding 32-bit elements of the two or four first source vectors and the indexed 32-bit element of the second source vector. The single-precision dot product results are destructively added to the corresponding single-precision elements of the ZA single-vector groups. The BF16 pairs within the second source vector are specified using an immediate index which selects the same BF16 pair position within each 128-bit vector segment. The element index range is from 0 to 3. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME2 ZA-targeting BFloat16 numerical behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

## Two ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	0	1	Zm			0	Rv		1	i2	Zn			0		1	1	off3				

BFDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 2;
```

## Four ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	0	1	Zm			1	Rv		1	i2	Zn			0	0	1	1	off3				

BFDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 4;
```

## Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	Is the immediate index of a group of two 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV 32;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m, VL];
  bits(VL) operand3 = ZAvector[vec, VL];
  for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(16) elt2_a = Elem[operand2, 2 * s + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * s + 1, 16];
    bits(32) sum = Elem[operand3, e, 32];
    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;
  ZAvector[vec, VL] = result;
vec = vec + vstride;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BFDOT (multiple and single vector)

Multi-vector BFloat16 floating-point dot-product by vector

The instruction computes the dot product of a pair of BF16 values held in the corresponding 32-bit elements of the two or four first source vectors and the second source vector. The single-precision dot product results are destructively added to the corresponding single-precision elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME2 ZA-targeting BFloat16 numerical behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

## Two ZA single-vectors (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			0	Rv		1	0	0	Zn			1	0	off3					

BFDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

## Four ZA single-vectors (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm			0	Rv		1	0	0	Zn			1	0	off3					

BFDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

## Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.



## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
  bits(VL) operand1 = Z[(n+r) MOD 32, VL];
  bits(VL) operand2 = Z[m, VL];
  bits(VL) operand3 = ZAVector[vec, VL];
  for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
    bits(32) sum = Elem[operand3, e, 32];
    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;
  ZAVector[vec, VL] = result;
vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## BFDOT (multiple vectors)

Multi-vector BFloat16 floating-point dot-product

The instruction computes the dot product of a pair of BF16 values held in the corresponding 32-bit elements of the two or four first and second source vectors. The single-precision dot product results are destructively added to the corresponding single-precision elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME2 ZA-targeting BFloat16 numerical behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

### Two ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	0	Rv		1	0	0	Zn			0	1	0	off3				

```
BFDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

### Four ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	1	0	Rv		1	0	0	Zn			0	0	1	0	off3		

```
BFDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

### Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.
- For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m+r, VL];
  bits(VL) operand3 = ZAvector[vec, VL];
  for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
    bits(32) sum = Elem[operand3, e, 32];
    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;
  ZAvector[vec, VL] = result;
  vec = vec + vstride;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFMAX (multiple and single vector)

Multi-vector BFloat16 floating-point maximum by vector

Determine the maximum of BFloat16 elements of the second source vector and the corresponding BFloat16 elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction follows SME2.1 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			1	0	1	0	0	0	0	1	0	0	0	Zdn			0		
size<1>size<0>																															

BFMAX { <Zdn1>.H-<Zdn2>.H }, { <Zdn1>.H-<Zdn2>.H }, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer dn = UInt(Zdn:'0');
integer m = UInt('0':Zm);
constant integer nreg = 2;
```

Four registers  
(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				1	0	1	0	1	0	0	0	1	0	0	0	Zdn		0	0
size<1>size<0>																															

BFMAX { <Zdn1>.H-<Zdn4>.H }, { <Zdn1>.H-<Zdn4>.H }, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer dn = UInt(Zdn:'00');
integer m = UInt('0':Zm);
constant integer nreg = 4;
```

Assembler Symbols

- <Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        bits(16) element1 = Elem[operand1, e, 16];
        bits(16) element2 = Elem[operand2, e, 16];
        Elem[results[r], e, 16] = BFMax(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BFMAX (multiple vectors)

Multi-vector BFloat16 floating-point maximum

Determine the maximum of BFloat16 elements of the two or four second source vectors and the corresponding BFloat16 elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction follows SME2.1 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

## Two registers

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	Zm				0	1	0	1	1	0	0	0	1	0	0	0	Zdn				0
size<1>size<0>																															

BFMAX { <Zdn1>.H-<Zdn2>.H }, { <Zdn1>.H-<Zdn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer dn = UInt(Zdn:'0');
integer m = UInt(Zm:'0');
constant integer nreg = 2;
```

## Four registers

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	Zm				0	0	1	0	1	1	1	0	0	1	0	0	0	Zdn		0	0
size<1>size<0>																															

BFMAX { <Zdn1>.H-<Zdn4>.H }, { <Zdn1>.H-<Zdn4>.H }, { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer dn = UInt(Zdn:'00');
integer m = UInt(Zm:'00');
constant integer nreg = 4;
```

## Assembler Symbols

- <Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.
- For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        bits(16) element1 = Elem[operand1, e, 16];
        bits(16) element2 = Elem[operand2, e, 16];
        Elem[results[r], e, 16] = BFMax(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BFMAXNM (multiple and single vector)

Multi-vector BFloat16 floating-point maximum number by vector

Determine the maximum number value of BFloat16 elements of the second source vector and the corresponding BFloat16 elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction follows SME2.1 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

## Two registers

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0		Zm		1	0	1	0	0	0	0	1	0	0	1		Zdn				0
size<1>size<0>																															

BFMAXNM { <Zdn1>.H-<Zdn2>.H }, { <Zdn1>.H-<Zdn2>.H }, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer dn = UInt(Zdn:'0');
integer m = UInt('0':Zm);
constant integer nreg = 2;
```

## Four registers

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			1	0	1	0	1	0	0	0	1	Zdn			0	0			
size<1>size<0>																															

BFMAXNM { <Zdn1>.H-<Zdn4>.H }, { <Zdn1>.H-<Zdn4>.H }, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer dn = UInt(Zdn:'00');
integer m = UInt('0':Zm);
constant integer nreg = 4;
```

## Assembler Symbols

- <Zdn1>

For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <Zdn4>

Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2>

Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.



## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        bits(16) element1 = Elem[operand1, e, 16];
        bits(16) element2 = Elem[operand2, e, 16];
        Elem[results[r], e, 16] = BFMaxNum(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BFMAXNM (multiple vectors)

Multi-vector BFloat16 floating-point maximum number

Determine the maximum number value of BFloat16 elements of the two or four second source vectors and the corresponding BFloat16 elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction follows SME2.1 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

## Two registers

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	Zm				0	1	0	1	1	0	0	0	1	0	0	1	Zdn				0
size<1>size<0>																															

BFMAXNM { <Zdn1>.H-<Zdn2>.H }, { <Zdn1>.H-<Zdn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer dn = UInt(Zdn:'0');
integer m = UInt(Zm:'0');
constant integer nreg = 2;
```

## Four registers

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	0	0	0	0	1	0	0	1	Zm				0	0	1	0	1	1	1	0	0	1	0	0	1	Zdn				0	0
size<1>size<0>																																	

BFMAXNM { <Zdn1>.H-<Zdn4>.H }, { <Zdn1>.H-<Zdn4>.H }, { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer dn = UInt(Zdn:'00');
integer m = UInt(Zm:'00');
constant integer nreg = 4;
```

## Assembler Symbols

- <Zdn1>

For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <Zdn4>

Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2>

Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.
- For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        bits(16) element1 = Elem[operand1, e, 16];
        bits(16) element2 = Elem[operand2, e, 16];
        Elem[results[r], e, 16] = BFMaxNum(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BFMIN (multiple and single vector)

Multi-vector BFloat16 floating-point minimum by vector

Determine the minimum of BFloat16 elements of the second source vector and the corresponding BFloat16 elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction follows SME2.1 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

## Two registers

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	0	0	1	0	Zm				1	0	1	0	0	0	0	1	0	0	0	0	Zdn				1
size<1>size<0>																																

BFMIN { <Zdn1>.H-<Zdn2>.H }, { <Zdn1>.H-<Zdn2>.H }, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer dn = UInt(Zdn:'0');
integer m = UInt('0':Zm);
constant integer nreg = 2;
```

## Four registers

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	0	0	0	0	1	0	0	1	0	Zm				1	0	1	0	1	0	0	0	1	0	0	0	Zdn				0	1
size<1>size<0>																																	

BFMIN { <Zdn1>.H-<Zdn4>.H }, { <Zdn1>.H-<Zdn4>.H }, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer dn = UInt(Zdn:'00');
integer m = UInt('0':Zm);
constant integer nreg = 4;
```

## Assembler Symbols

- <Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
  bits(VL) operand1 = Z[dn+r, VL];
  bits(VL) operand2 = Z[m, VL];
  for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, e, 16];
    bits(16) element2 = Elem[operand2, e, 16];
    Elem[results[r], e, 16] = BFMin(element1, element2, FPCR[]);

for r = 0 to nreg-1
  Z[dn+r, VL] = results[r];
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BFMIN (multiple vectors)

Multi-vector BFloat16 floating-point minimum

Determine the minimum of BFloat16 elements of the two or four second source vectors and the corresponding BFloat16 elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction follows SME2.1 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

## Two registers

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	Zm				0	1	0	1	1	0	0	0	1	0	0	0	Zdn				1
size<1>size<0>																															

BFMIN { <Zdn1>.H-<Zdn2>.H }, { <Zdn1>.H-<Zdn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer dn = UInt(Zdn:'0');
integer m = UInt(Zm:'0');
constant integer nreg = 2;
```

## Four registers

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	Zm				0	0	1	0	1	1	1	0	0	1	0	0	0	Zdn		0	1
size<1>size<0>																															

BFMIN { <Zdn1>.H-<Zdn4>.H }, { <Zdn1>.H-<Zdn4>.H }, { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer dn = UInt(Zdn:'00');
integer m = UInt(Zm:'00');
constant integer nreg = 4;
```

## Assembler Symbols

- <Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.
- For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        bits(16) element1 = Elem[operand1, e, 16];
        bits(16) element2 = Elem[operand2, e, 16];
        Elem[results[r], e, 16] = BFMin(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BFMINNM (multiple and single vector)

Multi-vector BFloat16 floating-point minimum number by vector

Determine the minimum number value of BFloat16 elements of the second source vector and the corresponding BFloat16 elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction follows SME2.1 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

## Two registers

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				1	0	1	0	0	0	0	1	0	0	1	Zdn			1	
size<1>size<0>																															

BFMINNM { <Zdn1>.H-<Zdn2>.H }, { <Zdn1>.H-<Zdn2>.H }, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer dn = UInt(Zdn:'0');
integer m = UInt('0':Zm);
constant integer nreg = 2;
```

## Four registers

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm				1	0	1	0	1	0	0	0	1	Zdn			0	1		
size<1>size<0>																															

BFMINNM { <Zdn1>.H-<Zdn4>.H }, { <Zdn1>.H-<Zdn4>.H }, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer dn = UInt(Zdn:'00');
integer m = UInt('0':Zm);
constant integer nreg = 4;
```

## Assembler Symbols

- <Zdn1>

For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <Zdn4>

Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2>

Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.



## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        bits(16) element1 = Elem[operand1, e, 16];
        bits(16) element2 = Elem[operand2, e, 16];
        Elem[results[r], e, 16] = BFMinNum(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BFMINNM (multiple vectors)

Multi-vector BFloat16 floating-point minimum number

Determine the minimum number value of BFloat16 elements of the two or four second source vectors and the corresponding BFloat16 elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.  
Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction follows SME2.1 non-widening BFloat16 numerical behaviors corresponding to instructions that place their results in two or four SVE Z vectors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

## Two registers

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	Zm				0	1	0	1	1	0	0	0	1	0	0	1	Zdn				1
size<1>size<0>																															

BFMINNM { <Zdn1>.H-<Zdn2>.H }, { <Zdn1>.H-<Zdn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer dn = UInt(Zdn:'0');
integer m = UInt(Zm:'0');
constant integer nreg = 2;
```

## Four registers

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	0	0	0	0	1	0	0	1	Zm				0	0	1	0	1	1	1	0	0	1	0	0	1	Zdn				0	1
size<1>size<0>																																	

BFMINNM { <Zdn1>.H-<Zdn4>.H }, { <Zdn1>.H-<Zdn4>.H }, { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer dn = UInt(Zdn:'00');
integer m = UInt(Zm:'00');
constant integer nreg = 4;
```

## Assembler Symbols

- <Zdn1>

For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <Zdn4>

Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2>

Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.
- For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        bits(16) element1 = Elem[operand1, e, 16];
        bits(16) element2 = Elem[operand2, e, 16];
        Elem[results[r], e, 16] = BFMInNum(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BFMLA (multiple and indexed vector)

Multi-vector BFloat16 floating-point fused multiply-add by indexed element

Multiply the indexed element of the second source vector by the corresponding BFloat16 floating-point elements of the two or four first source vectors and destructively add without intermediate rounding to the corresponding elements of the ZA single-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to 7, encoded in 3 bits. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME2.1 ZA-targeting non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

## Two ZA single-vectors

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				0	Rv		1	i3h		Zn				1	0	i3l	off3		
S																															

BFMLA ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i3h:i3l);
boolean sub_op = FALSE;
constant integer nreg = 2;
```

## Four ZA single-vectors

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				1	Rv		1	i3h		Zn				0	1	0	i3l	off3	
S																															

BFMLA ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i3h:i3l);
boolean sub_op = FALSE;
constant integer nreg = 4;
```

## Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1>	For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2. For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV 16;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m, VL];
  bits(VL) operand3 = ZAvector[vec, VL];
  for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, e, 16];
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(16) element2 = Elem[operand2, s, 16];
    bits(16) element3 = Elem[operand3, e, 16];
    if sub_op then element1 = BFNeg(element1);
    Elem[result, e, 16] = BFMulAdd_ZA(element3, element1, element2, FPCR[]);
  ZAvector[vec, VL] = result;
  vec = vec + vstride;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFMLA (multiple and single vector)

Multi-vector BFloat16 floating-point fused multiply-add by vector

Multiply the corresponding BFloat16 floating-point elements of the two or four first source vector with corresponding elements of the second source vector and destructively add without intermediate rounding to the corresponding elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME2.1 ZA-targeting non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

Two ZA single-vectors

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm			0	Rv		1	1	1	Zn			0		0	off3				
SZ																S															

BFMLA ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
boolean sub_op = FALSE;
constant integer nreg = 2;
```

Four ZA single-vectors

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	1	Zm			0	Rv		1	1	1	Zn			0	0	off3					
SZ																S															

BFMLA ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
boolean sub_op = FALSE;
constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
  bits(VL) operand1 = Z[(n+r) MOD 32, VL];
  bits(VL) operand2 = Z[m, VL];
  bits(VL) operand3 = ZAvector[vec, VL];
  for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, e, 16];
    bits(16) element2 = Elem[operand2, e, 16];
    bits(16) element3 = Elem[operand3, e, 16];
    if sub_op then element1 = BFNeg(element1);
    Elem[result, e, 16] = BFMulAdd_ZA(element3, element1, element2, FPCR[]);
  ZAvector[vec, VL] = result;
  vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## BFMLA (multiple vectors)

Multi-vector BFloat16 floating-point fused multiply-add

Multiply the corresponding BFloat16 floating-point elements of the two or four first and second source vectors and destructively add without intermediate rounding to the corresponding elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors. The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME2.1 ZA-targeting non-widening BFloat16 numerical behaviors. This instruction is unpredicated. ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented. It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

### Two ZA single-vectors (FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	Zm			0	0	Rv		1	0	0	Zn			0	0	1	off3				
SZ																S															

**BFMLA** ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
boolean sub_op = FALSE;
constant integer nreg = 2;
```

### Four ZA single-vectors (FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	Zm			0	1	0	Rv		1	0	0	Zn			0	0	0	1	off3		
SZ																S															

**BFMLA** ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
boolean sub_op = FALSE;
constant integer nreg = 4;
```

### Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.



- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.
- For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(16) element1 = Elem[operand1, e, 16];
        bits(16) element2 = Elem[operand2, e, 16];
        bits(16) element3 = Elem[operand3, e, 16];
        if sub_op then element1 = BFNeg(element1);
        Elem[result, e, 16] = BFMulAdd_ZA(element3, element1, element2, FPCR[]);
    ZAvector[vec, VL] = result;
    vec = vec + vstride;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BFMLAL (multiple and indexed vector)

Multi-vector BFloat16 floating-point multiply-add long by indexed element

This BFloat16 floating-point multiply-add long instruction widens all 16-bit BFloat16 elements in the one, two, or four first source vectors and the indexed element of the second source vector to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the ZA double-vector groups.

The BF16 elements within the second source vector are specified using a 3-bit immediate index which selects the same element position within each 128-bit vector segment.

The lowest of the two consecutive vector numbers forming the double-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: [One ZA double-vector](#) , [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

## One ZA double-vector

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	0	Zm			i3h	Rv	1	i3l	Zn			1	0	off3			S				

BFMLAL ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = FALSE;
constant integer nreg = 1;
```

## Two ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm			0	Rv	1	i3h	Zn			0	1	0	i3l	off2			S		

BFMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = FALSE;
constant integer nreg = 2;
```

## Four ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	0	0	1	Zm			1	Rv	1	i3h	Zn			0	0	1	0	i3l	off2			S		

```
BFMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = FALSE;
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
<offs2>	For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

## Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV 32;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 1
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer segmentbase = e - (e MOD eltspersegment);
      integer s = 2 * segmentbase + index;
      bits(16) element1 = Elem[operand1, 2 * e + i, 16];
      bits(16) element2 = Elem[operand2, s, 16];
      bits(32) element3 = Elem[operand3, e, 32];
      if sub_op then element1 = BFNeg(element1);
      Elem[result, e, 32] = BFMulAddH\_ZA(element3, element1, element2, FPCR[]);
      ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFMLAL (multiple and single vector)

Multi-vector BFloat16 floating-point multiply-add long by vector

This BFloat16 floating-point multiply-add long instruction widens all 16-bit BFloat16 elements in the one, two, or four first source vectors and the second source vector to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: [One ZA double-vector](#) , [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

One ZA double-vector  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			0	Rv		0	1	1	Zn			1	0	off3			S		

BFMLAL ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
boolean sub_op = FALSE;
constant integer nreg = 1;
```

Two ZA double-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			0	Rv		0	1	0	Zn			1	0	0	off2			S	

BFMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
boolean sub_op = FALSE;
constant integer nreg = 2;
```

Four ZA double-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm			0	Rv		0	1	0	Zn			1	0	0	off2			S	

```
BFMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
boolean sub_op = FALSE;
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
<offs2>	For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[(n+r) MOD 32, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 1
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      bits(16) element1 = Elem[operand1, 2 * e + i, 16];
      bits(16) element2 = Elem[operand2, 2 * e + i, 16];
      bits(32) element3 = Elem[operand3, e, 32];
      if sub_op then element1 = BFNeg(element1);
      Elem[result, e, 32] = BFMulAddH_ZA(element3, element1, element2, FPCR[]);
      ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## BFMLAL (multiple vectors)

Multi-vector BFloat16 floating-point multiply-add long

This BFloat16 floating-point multiply-add long instruction widens all 16-bit BFloat16 elements in the two or four first and second source vectors to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

### Two ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	0	Rv		0	1	0	Zn			0	1	0	0	0	off2		
S																															

BFMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off2:'0');
boolean sub_op = FALSE;
constant integer nreg = 2;
```

### Four ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	1	0	Rv		0	1	0	Zn			0	0	1	0	0	off2	
																															S

BFMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off2:'0');
boolean sub_op = FALSE;
constant integer nreg = 4;
```

### Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs1> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
- <offs2> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
- <Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  
For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m+r, VL];
  for i = 0 to 1
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      bits(16) element1 = Elem[operand1, 2 * e + i, 16];
      bits(16) element2 = Elem[operand2, 2 * e + i, 16];
      bits(32) element3 = Elem[operand3, e, 32];
      if sub_op then element1 = BFNeg(element1);
      Elem[result, e, 32] = BFMulAddH_ZA(element3, element1, element2, FPCR[]);
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## BFMLS (multiple and indexed vector)

Multi-vector BFloat16 floating-point fused multiply-subtract by indexed element

Multiply the indexed element of the second source vector by the corresponding BFloat16 floating-point elements of the two or four first source vectors and destructively subtract without intermediate rounding from the corresponding elements of the ZA single-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to 7, encoded in 3 bits. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME2.1 ZA-targeting non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

### Two ZA single-vectors

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				0	Rv		1	i3h		Zn				1	1	i3l	off3		
S																															

BFMLS ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i3h:i3l);
boolean sub_op = TRUE;
constant integer nreg = 2;
```

### Four ZA single-vectors

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				1	Rv		1	i3h		Zn				0	1	1	i3l	off3	
S																															

BFMLS ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i3h:i3l);
boolean sub_op = TRUE;
constant integer nreg = 4;
```

### Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1>	For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2. For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV 16;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m, VL];
  bits(VL) operand3 = ZAvector[vec, VL];
  for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, e, 16];
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(16) element2 = Elem[operand2, s, 16];
    bits(16) element3 = Elem[operand3, e, 16];
    if sub_op then element1 = BFNeg(element1);
    Elem[result, e, 16] = BFMulAdd_ZA(element3, element1, element2, FPCR[]);
  ZAvector[vec, VL] = result;
  vec = vec + vstride;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFMLS (multiple and single vector)

Multi-vector BFloat16 floating-point fused multiply-subtract by vector

Multiply the corresponding BFloat16 floating-point elements of the two or four first source vector with corresponding elements of the second source vector and destructively subtract without intermediate rounding from the corresponding elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME2.1 ZA-targeting non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

Two ZA single-vectors

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm			0	Rv		1	1	1	Zn			0	1	off3					
SZ																S															

BFMLS ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
boolean sub_op = TRUE;
constant integer nreg = 2;
```

Four ZA single-vectors

(FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	1	Zm			0	Rv		1	1	1	Zn			0	1	off3					
SZ																S															

BFMLS ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
boolean sub_op = TRUE;
constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(16) element1 = Elem[operand1, e, 16];
        bits(16) element2 = Elem[operand2, e, 16];
        bits(16) element3 = Elem[operand3, e, 16];
        if sub_op then element1 = BFNeg(element1);
        Elem[result, e, 16] = BFMulAdd_ZA(element3, element1, element2, FPCR[]);
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## BFMLS (multiple vectors)

Multi-vector BFloat16 floating-point fused multiply-subtract

Multiply the corresponding BFloat16 floating-point elements of the two or four first and second source vectors and destructively subtract without intermediate rounding from the corresponding elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors. The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME2.1 ZA-targeting non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

### Two ZA single-vectors (FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	Zm			0	0	Rv		1	0	0	Zn			0	1	1	off3				
SZ												S																			

**BFMLS** ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
boolean sub_op = TRUE;
constant integer nreg = 2;
```

### Four ZA single-vectors (FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	Zm			0	1	0	Rv		1	0	0	Zn			0	0	1	1	off3		
SZ												S																			

**BFMLS** ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
boolean sub_op = TRUE;
constant integer nreg = 4;
```

### Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.
- For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(16) element1 = Elem[operand1, e, 16];
        bits(16) element2 = Elem[operand2, e, 16];
        bits(16) element3 = Elem[operand3, e, 16];
        if sub_op then element1 = BFNeg(element1);
        Elem[result, e, 16] = BFMulAdd_ZA(element3, element1, element2, FPCR[]);
    ZAvector[vec, VL] = result;
    vec = vec + vstride;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFMLSL (multiple and indexed vector)

Multi-vector BFloat16 floating-point multiply-subtract long by indexed element

This BFloat16 floating-point multiply-subtract long instruction widens all 16-bit BFloat16 elements in the one, two, or four first source vectors and the indexed element of the second source vector to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the ZA double-vector groups.

The BF16 elements within the second source vector are specified using a 3-bit immediate index which selects the same element position within each 128-bit vector segment.

The lowest of the two consecutive vector numbers forming the double-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: [One ZA double-vector](#) , [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

One ZA double-vector  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	0	Zm			i3h	Rv	1	i3l	Zn			1	1	off3			S				

BFMLSL ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = TRUE;
constant integer nreg = 1;
```

Two ZA double-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm			0	Rv	1	i3h	Zn			0	1	1	i3l	off2			S		

BFMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = TRUE;
constant integer nreg = 2;
```

Four ZA double-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm			1	Rv	1	i3h	Zn			0	0	1	1	i3l	off2			S	

```
BFMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = TRUE;
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
<offs2>	For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.



## Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV 32;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 1
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer segmentbase = e - (e MOD eltspersegment);
      integer s = 2 * segmentbase + index;
      bits(16) element1 = Elem[operand1, 2 * e + i, 16];
      bits(16) element2 = Elem[operand2, s, 16];
      bits(32) element3 = Elem[operand3, e, 32];
      if sub_op then element1 = BFNeg(element1);
      Elem[result, e, 32] = BFMulAddH\_ZA(element3, element1, element2, FPCR[]);
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFMLSL (multiple and single vector)

Multi-vector BFloat16 floating-point multiply-subtract long by vector

This BFloat16 floating-point multiply-subtract long instruction widens all 16-bit BFloat16 elements in the one, two, or four first source vectors and the second source vector to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: [One ZA double-vector](#) , [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

One ZA double-vector

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			0	Rv		0	1	1	Zn			1	1	off3			S		

BFMLSL ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
boolean sub_op = TRUE;
constant integer nreg = 1;
```

Two ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			0	Rv		0	1	0	Zn			1	1	0	off2			S	

BFMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
boolean sub_op = TRUE;
constant integer nreg = 2;
```

Four ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm			0	Rv		0	1	0	Zn			1	1	0	off2			S	

```
BFMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
boolean sub_op = TRUE;
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
<offs2>	For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[(n+r) MOD 32, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 1
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      bits(16) element1 = Elem[operand1, 2 * e + i, 16];
      bits(16) element2 = Elem[operand2, 2 * e + i, 16];
      bits(32) element3 = Elem[operand3, e, 32];
      if sub_op then element1 = BFNeg(element1);
      Elem[result, e, 32] = BFMulAddH_ZA(element3, element1, element2, FPCR[]);
      ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFMLSL (multiple vectors)

Multi-vector BFloat16 floating-point multiply-subtract long

This BFloat16 floating-point multiply-subtract long instruction widens all 16-bit BFloat16 elements in the two or four first and second source vectors to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

Two ZA double-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm				0	0	Rv		0	1	0	Zn				0	1	1	0	off2	
S																															

BFMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off2:'0');
boolean sub_op = TRUE;
constant integer nreg = 2;
```

Four ZA double-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	1	0	Rv		0	1	0	Zn			0	0	1	1	0	off2	
S																															

BFMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off2:'0');
boolean sub_op = TRUE;
constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs1> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
- <offs2> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
- <Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  
For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m+r, VL];
  for i = 0 to 1
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      bits(16) element1 = Elem[operand1, 2 * e + i, 16];
      bits(16) element2 = Elem[operand2, 2 * e + i, 16];
      bits(32) element3 = Elem[operand3, e, 32];
      if sub_op then element1 = BFNeg(element1);
      Elem[result, e, 32] = BFMulAddH_ZA(element3, element1, element2, FPCR[]);
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

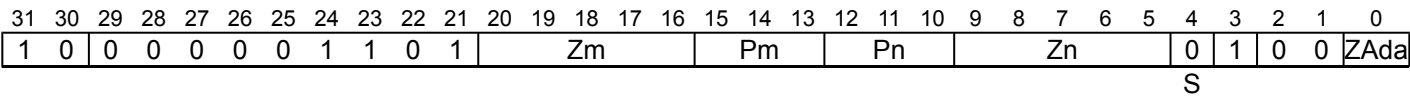
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFMOPA (non-widening)

BFloat16 floating-point outer product and accumulate

This instruction works with a 16-bit element ZA tile.  
These instructions generate an outer product of the first source vector and the second source vector. The first source is  $SVL_H \times 1$  vector and the second source is  $1 \times SVL_H$  vector.  
Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is Inactive the corresponding destination tile element remains unmodified.  
The resulting outer product,  $SVL_H \times SVL_H$ , is then destructively added to the destination tile. This is equivalent to performing a single multiply-accumulate to each of the destination tile elements.  
This instruction follows SME2.1 ZA-targeting non-widening BFloat16 numerical behaviors.  
ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented.

SME2  
(FEAT\_SVE\_B16B16)



BFMOPA <ZAda>.H, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
```

Assembler Symbols

- <ZAda> Is the name of the ZA tile ZA0-ZA1, encoded in the "ZAda" field.
- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV 16;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*16) operand3 = ZAtile[da, 16, dim*dim*16];
bits(dim*dim*16) result;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(16) element1 = Elem[operand1, row, 16];
        bits(16) element2 = Elem[operand2, col, 16];
        bits(16) element3 = Elem[operand3, row*dim+col, 16];

        if ActivePredicateElement(mask1, row, 16) && ActivePredicateElement(mask2, col, 16) then
            if sub_op then element1 = BFNeg(element1);
            Elem[result, row*dim+col, 16] = BFMulAdd\_ZA(element3, element1, element2, FPCR[]);
        else
            Elem[result, row*dim+col, 16] = element3;

ZAtile[da, 16, dim*dim*16] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFMOPA (widening)

BFloat16 sum of outer products and accumulate

The BFloat16 floating-point sum of outer products and accumulate instruction works with a 32-bit element ZA tile. This instruction multiplies the SVLs×2 sub-matrix of BFloat16 values held in the first source vector by the 2×SVLs sub-matrix of BFloat16 values in the second source vector.

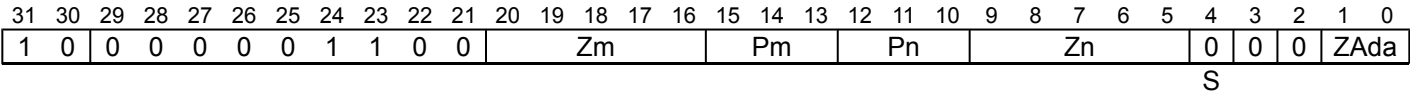
Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting SVLs×SVLs single-precision floating-point sum of outer products is then destructively added to the single-precision floating-point destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a SVLs×2 sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a 2×SVLs sub-matrix.

This instruction follows SME BFloat16 numerical behaviors.

SME
(FEAT\_SME)



BFMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSME() then UNDEFINED;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
```

Assembler Symbols

- <ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.



## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV 32;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
bits(dim*dim*32) result;

for row = 0 to dim-1
  for col = 0 to dim-1
    // determine row/col predicates
    boolean prow_0 = (ActivePredicateElement(mask1, 2*row + 0, 16));
    boolean prow_1 = (ActivePredicateElement(mask1, 2*row + 1, 16));
    boolean pcol_0 = (ActivePredicateElement(mask2, 2*col + 0, 16));
    boolean pcol_1 = (ActivePredicateElement(mask2, 2*col + 1, 16));

    bits(32) sum = Elem[operand3, row*dim+col, 32];
    if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
      bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0', 16));
      bits(16) erow_1 = (if prow_1 then Elem[operand1, 2*row + 1, 16] else FPZero('0', 16));
      bits(16) ecol_0 = (if pcol_0 then Elem[operand2, 2*col + 0, 16] else FPZero('0', 16));
      bits(16) ecol_1 = (if pcol_1 then Elem[operand2, 2*col + 1, 16] else FPZero('0', 16));
      if sub_op then
        boolean honor_alttp = FALSE; // Alternate handling ignored
        if prow_0 then erow_0 = BFNeg(erow_0, honor_alttp);
        if prow_1 then erow_1 = BFNeg(erow_1, honor_alttp);
        sum = BFDotAdd(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR[]);

    Elem[result, row*dim+col, 32] = sum;

ZAtile[da, 32, dim*dim*32] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

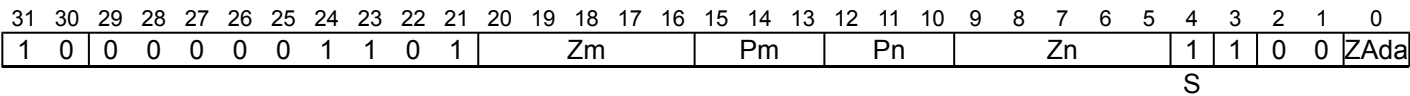
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFMOPS (non-widening)

BFloat16 floating-point outer product and subtract

This instruction works with a 16-bit element ZA tile.  
These instructions generate an outer product of the first source vector and the second source vector. The first source is  $SVL_H \times 1$  vector and the second source is  $1 \times SVL_H$  vector.  
Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is Inactive the corresponding destination tile element remains unmodified.  
The resulting outer product,  $SVL_H \times SVL_H$ , is then destructively subtracted from the destination tile. This is equivalent to performing a single multiply-subtract from each of the destination tile elements.  
This instruction follows SME2.1 ZA-targeting non-widening BFloat16 numerical behaviors.  
ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented.

SME2  
(FEAT\_SVE\_B16B16)



BFMOPS <ZAda>.H, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
```

Assembler Symbols

- <ZAda> Is the name of the ZA tile ZA0-ZA1, encoded in the "ZAda" field.
- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV 16;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*16) operand3 = ZAtile[da, 16, dim*dim*16];
bits(dim*dim*16) result;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(16) element1 = Elem[operand1, row, 16];
    bits(16) element2 = Elem[operand2, col, 16];
    bits(16) element3 = Elem[operand3, row*dim+col, 16];

    if ActivePredicateElement(mask1, row, 16) && ActivePredicateElement(mask2, col, 16) then
      if sub_op then element1 = BFNeg(element1);
      Elem[result, row*dim+col, 16] = BFMulAdd\_ZA(element3, element1, element2, FPCR[]);
    else
      Elem[result, row*dim+col, 16] = element3;

ZAtile[da, 16, dim*dim*16] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

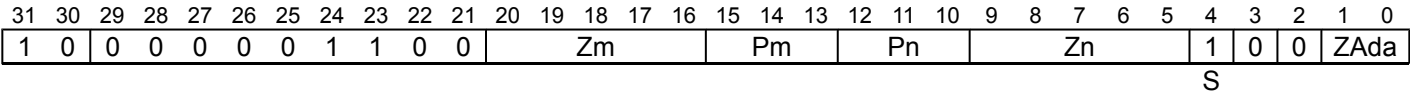
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFMOPS (widening)

BFloat16 sum of outer products and subtract

The BFloat16 floating-point sum of outer products and subtract instruction works with a 32-bit element ZA tile. This instruction multiplies the SVLs×2 sub-matrix of BFloat16 values held in the first source vector by the 2×SVLs sub-matrix of BFloat16 values in the second source vector. Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified. The resulting SVLs×SVLs single-precision floating-point sum of outer products is then destructively subtracted from the single-precision floating-point destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements. Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a SVLs×2 sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a 2×SVLs sub-matrix. This instruction follows SME BFloat16 numerical behaviors.

SME
(FEAT\_SME)



BFMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSME() then UNDEFINED;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
```

Assembler Symbols

- <ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV 32;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
bits(dim*dim*32) result;

for row = 0 to dim-1
  for col = 0 to dim-1
    // determine row/col predicates
    boolean prow_0 = (ActivePredicateElement(mask1, 2*row + 0, 16));
    boolean prow_1 = (ActivePredicateElement(mask1, 2*row + 1, 16));
    boolean pcol_0 = (ActivePredicateElement(mask2, 2*col + 0, 16));
    boolean pcol_1 = (ActivePredicateElement(mask2, 2*col + 1, 16));

    bits(32) sum = Elem[operand3, row*dim+col, 32];
    if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
      bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0', 16));
      bits(16) erow_1 = (if prow_1 then Elem[operand1, 2*row + 1, 16] else FPZero('0', 16));
      bits(16) ecol_0 = (if pcol_0 then Elem[operand2, 2*col + 0, 16] else FPZero('0', 16));
      bits(16) ecol_1 = (if pcol_1 then Elem[operand2, 2*col + 1, 16] else FPZero('0', 16));
      if sub_op then
        boolean honor_alttp = FALSE; // Alternate handling ignored
        if prow_0 then erow_0 = BFNeg(erow_0, honor_alttp);
        if prow_1 then erow_1 = BFNeg(erow_1, honor_alttp);
        sum = BFDotAdd(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR[]);

    Elem[result, row*dim+col, 32] = sum;

ZAtile[da, 32, dim*dim*32] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# BFSUB

BFloat16 floating-point subtract multi-vector from ZA array vector accumulators

Destructively subtract all elements of the two or four source vectors from the corresponding BFloat16 elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME2.1 ZA-targeting non-widening BFloat16 numerical behaviors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.B16B16 indicates whether this instruction is implemented.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

## Two ZA single-vectors (FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	0	0	1	0	0	0	Rv	1	1	1	Zm			0	0	1	off3				
SZ																S															

BFSUB ZA.H[<Wv>, <offs>{, VGx2}], { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer v = UInt('010':Rv);
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

## Four ZA single-vectors (FEAT\_SVE\_B16B16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	0	0	1	0	1	0	Rv	1	1	1	Zm			0	0	0	1	off3			
SZ																S															

BFSUB ZA.H[<Wv>, <offs>{, VGx4}], { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SVE_B16B16) then UNDEFINED;
integer v = UInt('010':Rv);
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

## Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  
For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
  bits(VL) operand1 = ZAvector[vec, VL];
  bits(VL) operand2 = Z[m+r, VL];
  for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, e, 16];
    bits(16) element2 = Elem[operand2, e, 16];
    Elem[result, e, 16] = BFSUB\_ZA(element1, element2, FPCR[]);
  ZAvector[vec, VL] = result;
  vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFVDDOT

Multi-vector BFloat16 floating-point vertical dot-product by indexed element

The instruction computes the sum-of-products of each vertical pair of BFloat16 values in the corresponding elements of the two first source vectors with the pair of BFloat16 values in the indexed 32-bit group of the corresponding 128-bit segment of the second source vector. The single-precision sum-of-products results are destructively added to the corresponding single-precision elements of the two ZA single-vector groups.

The BF16 pairs within the second source vector are specified using an immediate index which selects the same BF16 pair position within each 128-bit vector segment. The element index range is from 0 to 3.

The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

The VECTOR GROUP symbol VGx2 indicates that the ZA operand consists of two ZA single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME2 ZA-targeting BFloat16 numerical behaviors.

This instruction is unpredicated.

SME2  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				0	Rv		0	i2		Zn				0	1	1	off3		

BFVDDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> Is the immediate index of a group of two 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.



## Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV 2;
integer eltspersegment = 128 DIV 32;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to 1
    bits(VL) operand1a = Z[n, VL];
    bits(VL) operand1b = Z[n+1, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        bits(16) elt1_a = Elem[operand1a, 2 * e + r, 16];
        bits(16) elt1_b = Elem[operand1b, 2 * e + r, 16];
        bits(16) elt2_a = Elem[operand2, 2 * s + 0, 16];
        bits(16) elt2_b = Elem[operand2, 2 * s + 1, 16];
        bits(32) sum = Elem[operand3, e, 32];
        sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
        Elem[result, e, 32] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

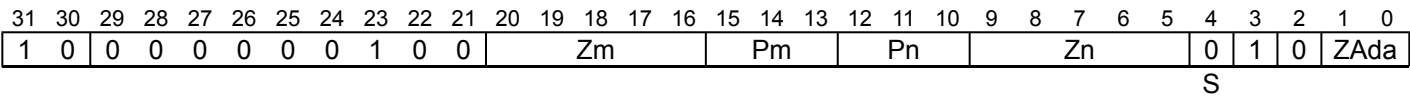
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BMOPA

Bitwise exclusive NOR population count outer product and accumulate

This instruction works with 32-bit element ZA tile. This instruction generates an outer product of the first source  $SVL_S \times 1$  vector and the second source  $1 \times SVL_S$  vector. Each outer product element is obtained as population count of the bitwise XNOR result of the corresponding 32-bit elements of the first source vector and the second source vector. Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is inactive the corresponding destination tile element remains unmodified. The resulting  $SVL_S \times SVL_S$  product is then destructively added to the destination tile.

SME2  
(FEAT\_SME2)



BMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S, <Zm>.S

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
```

Assembler Symbols

- <ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for row = 0 to dim-1
  bits(esize) element1 = Elem[operand1, row, esize];
  for col = 0 to dim-1
    bits(esize) element2 = Elem[operand2, col, esize];
    bits(esize) element3 = Elem[operand3, row*dim + col, esize];
    if (ActivePredicateElement(mask1, row, esize) &&
        ActivePredicateElement(mask2, col, esize)) then
      integer res = BitCount(NOT(element1 EOR element2));
      if sub_op then res = -res;
      Elem[result, row*dim + col, esize] = element3 + res;
    else
      Elem[result, row*dim + col, esize] = element3;
  ZAtile[da, esize, dim*dim*esize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

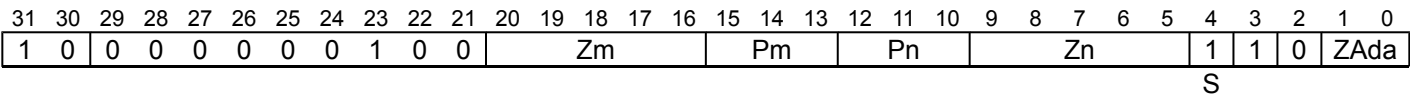
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BMOPS

Bitwise exclusive NOR population count outer product and subtract

This instruction works with 32-bit element ZA tile. This instruction generates an outer product of the first source  $SVL_S \times 1$  vector and the second source  $1 \times SVL_S$  vector. Each outer product element is obtained as population count of the bitwise XNOR result of the corresponding 32-bit elements of the first source vector and the second source vector. Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is inactive the corresponding destination tile element remains unmodified. The resulting  $SVL_S \times SVL_S$  product is then destructively subtracted from the destination tile.

SME2  
(FEAT\_SME2)



BMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S, <Zm>.S

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
```

Assembler Symbols

- <ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for row = 0 to dim-1
  bits(esize) element1 = Elem[operand1, row, esize];
  for col = 0 to dim-1
    bits(esize) element2 = Elem[operand2, col, esize];
    bits(esize) element3 = Elem[operand3, row*dim + col, esize];
    if (ActivePredicateElement(mask1, row, esize) &&
        ActivePredicateElement(mask2, col, esize)) then
      integer res = BitCount(NOT(element1 EOR element2));
      if sub_op then res = -res;
      Elem[result, row*dim + col, esize] = element3 + res;
    else
      Elem[result, row*dim + col, esize] = element3;
  ZAtile[da, esize, dim*dim*esize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FADD

Floating-point add multi-vector to ZA array vector accumulators

Destructively add all elements of the two or four source vectors to the corresponding elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.F64F64 indicates whether the double-precision variant is implemented, and

ID\_AA64SMFR0\_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 4 classes: [Two ZA single-vectors](#) , [Two ZA single-vectors of half precision elements](#) , [Four ZA single-vectors](#) and [Four ZA single-vectors of half precision elements](#)

Two ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	0	0	Rv	1	1	1	Zm			0	0	0	off3				
S																															

FADD ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Two ZA single-vectors of half precision elements

(FEAT\_SME\_F16F16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	0	0	1	0	0	0	Rv	1	1	1	Zm			0	0	0	off3				
SZ										S																					

FADD ZA.H[<Wv>, <offs>{, VGx2}], { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors

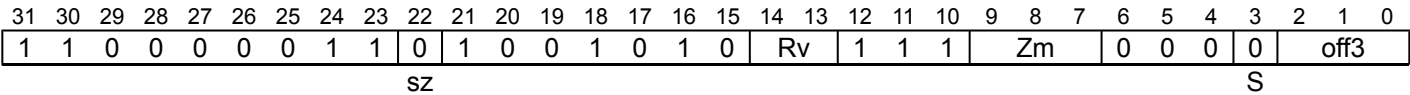
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	1	0	Rv	1	1	1	Zm			0			0	0	0	off3		
S																																

```
FADD ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zm1>.<T>-<Zm4>.<T> }
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Four ZA single-vectors of half precision elements  
(FEAT\_SME\_F16F16)



```
FADD ZA.H[<Wv>, <offs>{, VGx4}], { <Zm1>.H-<Zm4>.H }
```

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zm1> For the two ZA single-vectors and two ZA single-vectors of half precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  
For the four ZA single-vectors and four ZA single-vectors of half precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
  bits(VL) operand1 = ZAvector[vec, VL];
  bits(VL) operand2 = Z[m+r, VL];
  for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd\_ZA(element1, element2, FPCR[]);
  ZAvector[vec, VL] = result;
  vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



FCLAMP

Multi-vector floating-point clamp to minimum/maximum number

Clamp each floating-point element in the two or four destination vectors to between the floating-point minimum value in the corresponding element of the first source vector and the floating-point maximum value in the corresponding element of the second source vector and destructively place the clamped results in the corresponding elements of the two or four destination vectors.

Regardless of the value of FPCR.AH, the behavior is as follows for each mininum number and maximum number operation:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	!= 00	1	Zm						1	1	0	0	0	0	Zn				Zd			0		
size																															

FCLAMP { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<T>, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd:'0');
constant integer nreg = 2;
```

Four registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	!= 00	1	Zm						1	1	0	0	1	0	Zn				Zd			0	0	
size																															

FCLAMP { <Zd1>.<T>-<Zd4>.<T> }, <Zn>.<T>, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd:'00');
constant integer nreg = 4;
```

Assembler Symbols

- <Zd1>
- For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.  
  
For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <T>
- Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = Z[d+r, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];
        Elem[results[r], e, esize] = FPMinNum(FPMaxNum(element1, element3, FPCR[]), element2, FPCR[]);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# FCVT (narrowing)

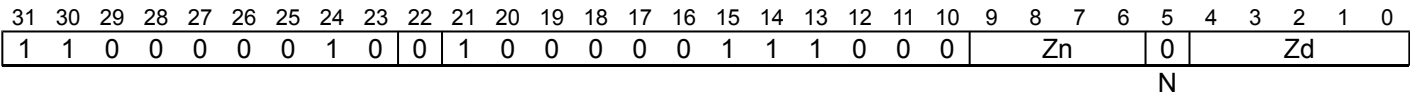
Multi-vector floating-point convert from single-precision to packed half-precision

Convert to half-precision from single-precision, each element of the two source vectors, and place the results in the half-width destination elements.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

## SME2 (FEAT\_SME2)



FCVT <Zd>.H, { <Zn1>.S-<Zn2>.S }

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

## Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
bits(VL) result;

bits(VL) operand1 = Z[n+0, VL];
bits(VL) operand2 = Z[n+1, VL];
for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, e, 32];
    bits(32) element2 = Elem[operand2, e, 32];
    bits(16) res1 = FPConvertSVE(element1, FPCR[], 16);
    bits(16) res2 = FPConvertSVE(element2, FPCR[], 16);
    Elem[result, e, 16] = res1;
    Elem[result, elements+e, 16] = res2;

Z[d, VL] = result;
```

FCVT (widening)

Multi-vector floating-point convert from half-precision to single-precision (in-order)

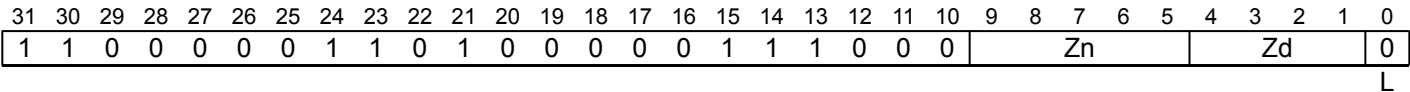
Convert to single-precision from half-precision, each element of the source vector, and place the results in the double-width destination elements of the destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.F16F16 indicates whether this instruction is implemented.

SME2  
(FEAT\_SME\_F16F16)



FCVT { <Zd1>.S-<Zd2>.S }, <Zn>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd:'0');
```

Assembler Symbols

- <Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 16;
bits(VL) operand = Z[n, VL];
bits(2*VL) result;

for e = 0 to elements-1
    bits(16) element = Elem[operand, e, 16];
    bits(32) res = FPConvertSVE(element, FPCR[], 32);
    Elem[result, e, 32] = res;

Z[d+0, VL] = result<VL-1:0>;
Z[d+1, VL] = result<2*VL-1:VL>;
```

FCVTL

Multi-vector floating-point convert from half-precision to deinterleaved single-precision

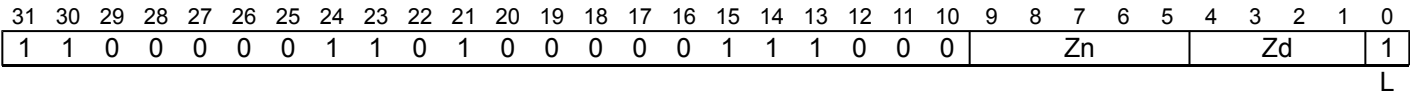
Convert to single-precision from half-precision, each element of the source vector, and place the deinterleaved results in the double-width destination elements of the destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.F16F16 indicates whether this instruction is implemented.

SME2  
(FEAT\_SME\_F16F16)



FCVTL { <Zd1>.S-<Zd2>.S }, <Zn>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd:'0');
```

Assembler Symbols

- <Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer pairs = VL DIV 32;
bits(VL) operand = Z[n, VL];
bits(VL) result0;
bits(VL) result1;

for p = 0 to pairs-1
    bits(16) element1 = Elem[operand, 2*p+0, 16];
    bits(16) element2 = Elem[operand, 2*p+1, 16];
    bits(32) res1 = FPConvertSVE(element1, FPCR[], 32);
    bits(32) res2 = FPConvertSVE(element2, FPCR[], 32);
    Elem[result0, p, 32] = res1;
    Elem[result1, p, 32] = res2;

Z[d+0, VL] = result0;
Z[d+1, VL] = result1;
```

FCVTN

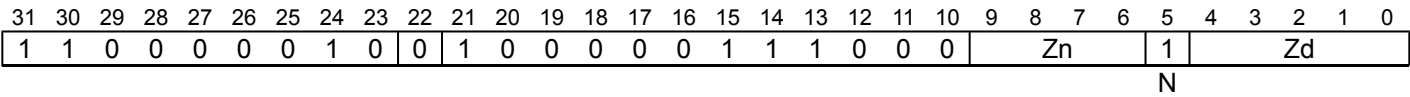
Multi-vector floating-point convert from single-precision to interleaved half-precision

Convert to half-precision from single-precision, each element of the two source vectors, and place the two-way interleaved results in the half-width destination elements.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

SME2  
(FEAT\_SME2)



FCVTN <Zd>.H, { <Zn1>.S-<Zn2>.S }

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
bits(VL) result;

bits(VL) operand1 = Z[n+0, VL];
bits(VL) operand2 = Z[n+1, VL];
for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, e, 32];
    bits(32) element2 = Elem[operand2, e, 32];
    bits(16) res1 = FPConvertSVE(element1, FPCR[], 16);
    bits(16) res2 = FPConvertSVE(element2, FPCR[], 16);
    Elem[result, 2*e + 0, 16] = res1;
    Elem[result, 2*e + 1, 16] = res2;

Z[d, VL] = result;
```

FCVTZS

Multi-vector floating-point convert to signed integer, rounding toward zero

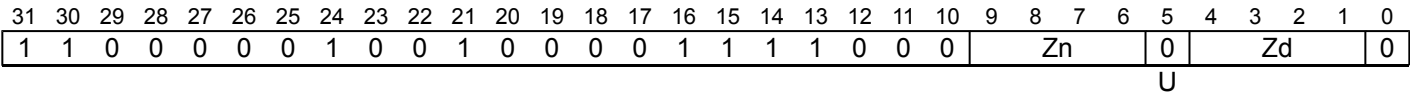
Convert to the signed 32-bit integer nearer to zero from single-precision, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

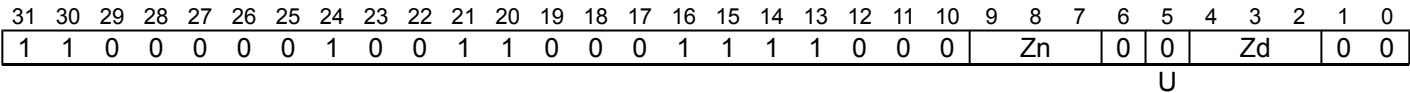
Two registers  
(FEAT\_SME2)



```
FCVTZS { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd:'0');
constant integer nreg = 2;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

Four registers  
(FEAT\_SME2)



```
FCVTZS { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'00');
integer d = UInt(Zd:'00');
constant integer nreg = 4;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

Assembler Symbols

- <Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.  
For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(32) element = Elem[operand, e, 32];
        Elem[results[r], e, 32] = FPToFixed(element, 0, unsigned, FPCR[], rounding, 32);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



FCVTZU

Multi-vector floating-point convert to unsigned integer, rounding toward zero

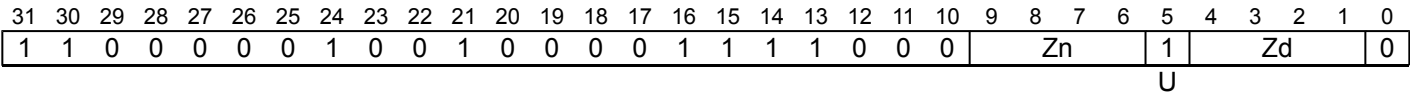
Convert to the unsigned 32-bit integer nearer to zero from single-precision, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

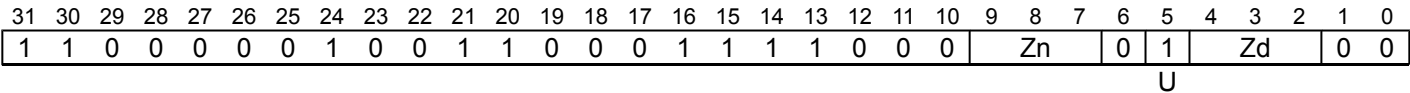
Two registers  
(FEAT\_SME2)



```
FCVTZU { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd:'0');
constant integer nreg = 2;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

Four registers  
(FEAT\_SME2)



```
FCVTZU { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'00');
integer d = UInt(Zd:'00');
constant integer nreg = 4;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

Assembler Symbols

- <Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.  
For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(32) element = Elem[operand, e, 32];
        Elem[results[r], e, 32] = FPToFixed(element, 0, unsigned, FPCR[], rounding, 32);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# FDOT (multiple and indexed vector)

Multi-vector half-precision floating-point dot-product by indexed element

The instruction computes the fused sum-of-products of a pair of half-precision floating-point values held in the corresponding 32-bit elements of the two or four first source vectors and the indexed 32-bit element of the second source vector, without intermediate rounding. The single-precision sum-of-products results are destructively added to the corresponding single-precision elements of the ZA single-vector groups.

The half-precision floating-point pairs within the second source vector are specified using an immediate index which selects the same pair position within each 128-bit vector segment. The element index range is from 0 to 3. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

## Two ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	0	1	Zm			0	Rv		1	i2		Zn			0		0	1	off3			

FDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 2;
```

## Four ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	0	1		Zm		1		Rv		1		i2		Zn		0	0	0	1		off3	

FDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 4;
```

## Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	Is the immediate index of a group of two 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

## Operation

```

CheckStreamingSVEAndZAEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV 32;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m, VL];
  bits(VL) operand3 = ZAvector[vec, VL];
  for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(16) elt2_a = Elem[operand2, 2 * s + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * s + 1, 16];
    bits(32) sum = Elem[operand3, e, 32];
    sum = FPDotAdd_ZA(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;
  ZAvector[vec, VL] = result;
  vec = vec + vstride;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FDOT (multiple and single vector)

Multi-vector half-precision floating-point dot-product by vector

The instruction computes the fused sum-of-products of a pair of half-precision floating-point values held in the corresponding 32-bit elements of the two or four first source vectors and the second source vector, without intermediate rounding. The single-precision sum-of-products results are destructively added to the corresponding single-precision elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

### Two ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			0	Rv		1			0	0	Zn			0		0	off3		

FDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

### Four ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm			0	Rv		1			0	0	Zn			0		0	off3		

FDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

### Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
  bits(VL) operand1 = Z[(n+r) MOD 32, VL];
  bits(VL) operand2 = Z[m, VL];
  bits(VL) operand3 = ZAVector[vec, VL];
  for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
    bits(32) sum = Elem[operand3, e, 32];
    sum = FPDotAdd\_ZA(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;
  ZAVector[vec, VL] = result;
vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FDOT (multiple vectors)

Multi-vector half-precision floating-point dot-product

The instruction computes the fused sum-of-products of a pair of half-precision floating-point values held in the corresponding 32-bit elements of the two or four first and second source vectors, without intermediate rounding. The single-precision sum-of-products results are destructively added to the corresponding single-precision elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

Two ZA single-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	0	Rv		1	0	0	Zn			0	0	0	off3				

FDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	1	0	Rv		1	0	0	Zn			0	0	0	0	off3		

FDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.
- For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m+r, VL];
  bits(VL) operand3 = ZAvector[vec, VL];
  for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
    bits(32) sum = Elem[operand3, e, 32];
    sum = FPDotAdd_ZA(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;
  ZAvector[vec, VL] = result;
  vec = vec + vstride;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



FMAX (multiple and single vector)

Multi-vector floating-point maximum by vector

Determine the maximum of floating-point elements of the second source vector and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	!= 00	1	0	Zm					1	0	1	0	0	0	0	1	0	0	0	0	Zdn			0
size																															

FMAX { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt('0':Zm);
constant integer nreg = 2;
```

Four registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	!= 00	1	0	Zm					1	0	1	0	1	0	0	1	0	0	0	0	Zdn		0	0
size																															

FMAX { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt('0':Zm);
constant integer nreg = 4;
```

Assembler Symbols

- <Zdn1>
- For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <T>
- Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[results[r], e, esize] = FPMAX(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMAX (multiple vectors)

Multi-vector floating-point maximum

Determine the maximum of floating-point elements of the two or four second source vectors and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	!= 00	1	Zm				0	1	0	1	1	0	0	0	1	0	0	0	0	Zdn				0
size																															

FMAX { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt(Zm:'0');
constant integer nreg = 2;
```

Four registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	!= 00	1	Zm				0	0	1	0	1	1	1	0	0	1	0	0	0	Zdn				0	0
size																																

FMAX { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt(Zm:'00');
constant integer nreg = 4;
```

Assembler Symbols

- <Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <T> Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[results[r], e, esize] = FPMax(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMAXNM (multiple and single vector)

Multi-vector floating-point maximum number by vector

Determine the maximum number value of floating-point elements of the second source vector and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	!= 00	1	0	Zm					1	0	1	0	0	0	0	1	0	0	1	Zdn				0
size																															

**FMAXNM** { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt('0':Zm);
constant integer nreg = 2;
```

### Four registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	!= 00	1	0	Zm					1	0	1	0	1	0	0	1	0	0	1	Zdn			0	0
size																															

**FMAXNM** { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt('0':Zm);
constant integer nreg = 4;
```

### Assembler Symbols

- <Zdn1>      For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.
- For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <T>         Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[results[r], e, esize] = FPCMaxNum(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# FMAXNM (multiple vectors)

Multi-vector floating-point maximum number

Determine the maximum number value of floating-point elements of the two or four second source vectors and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.  
Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.  
This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

## Two registers (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	!= 00	1		Zm		0	1	0	1	1	0	0	0	1	0	0	1		Zdn					0
size																															

FMAXNM { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt(Zm:'0');
constant integer nreg = 2;
```

## Four registers (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	!= 00	1		Zm		0	0	1	0	1	1	1	0	0	1	0	0	1		Zdn		0	0	
size																															

FMAXNM { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt(Zm:'00');
constant integer nreg = 4;
```

## Assembler Symbols

- <Zdn1>
- For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <T>
- Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[results[r], e, esize] = FPMaxNum(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



FMIN (multiple and single vector)

Multi-vector floating-point minimum by vector

Determine the minimum of floating-point elements of the second source vector and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	!= 00	1	0	Zm				1	0	1	0	0	0	0	1	0	0	0	Zdn				1	
size																															

FMIN { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt('0':Zm);
constant integer nreg = 2;
```

Four registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	!= 00	1	0	Zm				1	0	1	0	1	0	0	1	0	0	0	0	Zdn			0	1
size																															

FMIN { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt('0':Zm);
constant integer nreg = 4;
```

Assembler Symbols

- <Zdn1>
- For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <T>
- Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[results[r], e, esize] = FPMIn(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# FMIN (multiple vectors)

Multi-vector floating-point minimum

Determine the minimum of floating-point elements of the two or four second source vectors and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

## Two registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	!= 00	1	Zm					0	1	0	1	1	0	0	0	1	0	0	0	Zdn					1
size																																

FMIN { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt(Zm:'0');
constant integer nreg = 2;
```

## Four registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	!= 00	1	Zm				0	0	1	0	1	1	1	0	0	1	0	0	0	Zdn			0	1
size																															

FMIN { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt(Zm:'00');
constant integer nreg = 4;
```

## Assembler Symbols

- <Zdn1>

For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <T>

Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[results[r], e, esize] = FPMIn(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# FMINNM (multiple and single vector)

Multi-vector floating-point minimum number by vector

Determine the minimum number value of floating-point elements of the second source vector and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

## Two registers (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	!= 00	1	0	Zm					1	0	1	0	0	0	0	0	1	0	0	1	Zdn			1
size																															

FMINNM { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt('0':Zm);
constant integer nreg = 2;
```

## Four registers (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	!= 00	1	0		Zm		1	0	1	0	1	0	0	0	0	1	0	0	1		Zdn		0	1
size																															

FMINNM { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt('0':Zm);
constant integer nreg = 4;
```

## Assembler Symbols

- <Zdn1>
- For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <T>
- Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[results[r], e, esize] = FPMinNum(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# FMINNM (multiple vectors)

Multi-vector floating-point minimum number

Determine the minimum number value of floating-point elements of the two or four second source vectors and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

## Two registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	!= 00	1	Zm			0	1	0	1	1	0	0	0	1	0	0	1	Zdn			1			
size																															

FMINNM { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt(Zm:'0');
constant integer nreg = 2;
```

## Four registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	!= 00	1	Zm			0	0	1	0	1	1	1	0	0	1	0	0	1	Zdn			0	1	
size																															

FMINNM { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt(Zm:'00');
constant integer nreg = 4;
```

## Assembler Symbols

- <Zdn1>
- For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <T>
- Is the size specifier, encoded in "size":

size	<T>
01	H
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[results[r], e, esize] = FPMInNum(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## FMLA (multiple and indexed vector)

Multi-vector floating-point fused multiply-add by indexed element

Multiply the indexed element of the second source vector by the corresponding floating-point elements of the two or four first source vectors and destructively add without intermediate rounding to the corresponding elements of the ZA single-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 2 bits depending on the size of the element. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.F64F64 indicates whether the double-precision variant is implemented, and

ID\_AA64SMFR0\_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 6 classes: [Two ZA single-vectors of half precision elements](#) , [Two ZA single-vectors of single precision elements](#) , [Two ZA single-vectors of double precision elements](#) , [Four ZA single-vectors of half precision elements](#) , [Four ZA single-vectors of single precision elements](#) and [Four ZA single-vectors of double precision elements](#)

### Two ZA single-vectors of half precision elements

(FEAT\_SME\_F16F16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				0	Rv		1	i3h		Zn				0	0	i3l		off3	
S																															

**FMLA** ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i3h:i3l);
boolean sub_op = FALSE;
constant integer nreg = 2;
```

### Two ZA single-vectors of single precision elements

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				0	Rv		0	i2		Zn				0	0	0	off3		
S																															

**FMLA** ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.S-<Zn2>.S }, <Zm>.S[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
boolean sub_op = FALSE;
constant integer nreg = 2;
```

## Two ZA single-vectors of double precision elements

(FEAT\_SME\_F64F64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1		Zm		0		Rv		0	0	i1		Zn		0	0	0		off3		
																															S

```
FMLA ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.D-<Zn2>.D }, <Zm>.D[<index>]
```

```
if !(HaveSME2() && HaveSMEF64F64()) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 64;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i1);
boolean sub_op = FALSE;
constant integer nreg = 2;
```

## Four ZA single-vectors of half precision elements

(FEAT\_SME\_F16F16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1		Zm		1		Rv		1	i3h		Zn		0	0	0	i3l		off3		
																															S

```
FMLA ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i3h:i3l);
boolean sub_op = FALSE;
constant integer nreg = 4;
```

## Four ZA single-vectors of single precision elements

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	0	1		Zm		1		Rv		0	i2		Zn		0	0	0	0		off3		
																															S

```
FMLA ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.S-<Zn4>.S }, <Zm>.S[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
boolean sub_op = FALSE;
constant integer nreg = 4;
```

## Four ZA single-vectors of double precision elements

(FEAT\_SME\_F64F64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm			1	Rv		0	0	i1	Zn			0	0	0	0	off3			
																												S			

```
FMLA ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.D-<Zn4>.D }, <Zm>.D[<index>]
```

```
if !(HaveSME2() && HaveSMEF64F64()) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 64;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i1);
boolean sub_op = FALSE;
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs>	Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
<Zn1>	For the two ZA single-vectors of double precision elements, two ZA single-vectors of half precision elements and two ZA single-vectors of single precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA single-vectors of double precision elements, four ZA single-vectors of half precision elements and four ZA single-vectors of single precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	For the four ZA single-vectors of half precision elements and two ZA single-vectors of half precision elements variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  For the four ZA single-vectors of single precision elements and two ZA single-vectors of single precision elements variant: is the element index, in the range 0 to 3, encoded in the "i2" field.  For the four ZA single-vectors of double precision elements and two ZA single-vectors of double precision elements variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltsperssegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        integer segmentbase = e - (e MOD eltsperssegment);
        integer s = segmentbase + index;
        bits(esize) element2 = Elem[operand2, s, esize];
        bits(esize) element3 = Elem[operand3, e, esize];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAdd_ZA(element3, element1, element2, FPCR[]);
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```



## FMLA (multiple and single vector)

Multi-vector floating-point fused multiply-add by vector

Multiply the corresponding floating-point elements of the two or four first source vector with corresponding elements of the second source vector and destructively add without intermediate rounding to the corresponding elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.F64F64 indicates whether the double-precision variant is implemented, and

ID\_AA64SMFR0\_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 4 classes: [Two ZA single-vectors](#) , [Two ZA single-vectors of half precision elements](#) , [Four ZA single-vectors](#) and [Four ZA single-vectors of half precision elements](#)

### Two ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0		Zm		0		Rv	1	1	0			Zn		0	0		off3			
																S															

**FMLA** ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
boolean sub_op = FALSE;
constant integer nreg = 2;
```

### Two ZA single-vectors of half precision elements

(FEAT\_SME\_F16F16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			0	Rv		1	1	1	Zn			0	0	off3					
SZ										S																					

**FMLA** ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
boolean sub_op = FALSE;
constant integer nreg = 2;
```

### Four ZA single-vectors

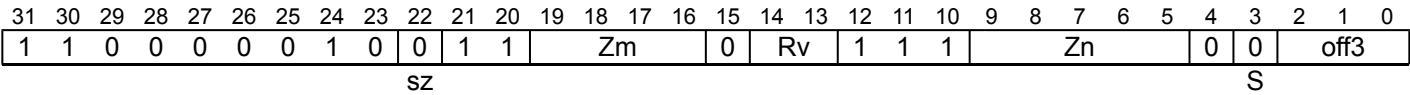
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm		0		Rv		1	1	0	Zn			0		0	off3				
																S															

FMLA ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
boolean sub_op = FALSE;
constant integer nreg = 4;
```

Four ZA single-vectors of half precision elements  
(FEAT\_SME\_F16F16)



FMLA ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
boolean sub_op = FALSE;
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
  bits(VL) operand1 = Z[(n+r) MOD 32, VL];
  bits(VL) operand2 = Z[m, VL];
  bits(VL) operand3 = ZAvector[vec, VL];
  for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    bits(esize) element3 = Elem[operand3, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd\_ZA(element3, element1, element2, FPCR[]);
  ZAvector[vec, VL] = result;
  vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMLA (multiple vectors)

Multi-vector floating-point fused multiply-add

Multiply the corresponding floating-point elements of the two or four first and second source vectors and destructively add without intermediate rounding to the corresponding elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors. The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.F64F64 indicates whether the double-precision variant is implemented, and ID\_AA64SMFR0\_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 4 classes: [Two ZA single-vectors](#) , [Two ZA single-vectors of half precision elements](#) , [Four ZA single-vectors](#) and [Four ZA single-vectors of half precision elements](#)

Two ZA single-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1			Zm		0	0		Rv	1	1	0			Zn		0	0	0		off3	
																S															

FMLA ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
boolean sub_op = FALSE;
constant integer nreg = 2;
```

Two ZA single-vectors of half precision elements  
(FEAT\_SME\_F16F16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1			Zm		0	0		Rv	1	0	0			Zn		0	0	1		off3	
SZ										S																					

FMLA ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
boolean sub_op = FALSE;
constant integer nreg = 2;
```

Four ZA single-vectors  
(FEAT\_SME2)

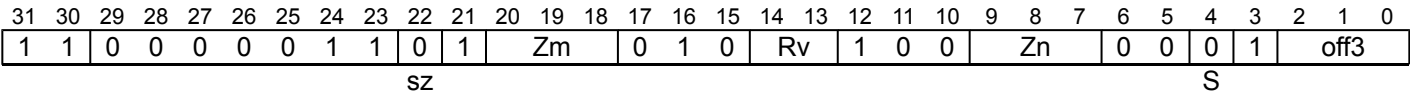
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm		0		1	0	Rv		1	1	0	Zn		0		0	0	0	off3		
S																															



FMLA ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
boolean sub_op = FALSE;
constant integer nreg = 4;
```

Four ZA single-vectors of half precision elements  
(FEAT\_SME\_F16F16)



FMLA ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
boolean sub_op = FALSE;
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors and two ZA single-vectors of half precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA single-vectors and four ZA single-vectors of half precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA single-vectors and two ZA single-vectors of half precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  
For the four ZA single-vectors and four ZA single-vectors of half precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m+r, VL];
  bits(VL) operand3 = ZAvector[vec, VL];
  for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    bits(esize) element3 = Elem[operand3, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd\_ZA(element3, element1, element2, FPCR[]);
  ZAvector[vec, VL] = result;
vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMLAL (multiple and indexed vector)

Multi-vector floating-point multiply-add long by indexed element

This half-precision floating-point multiply-add long instruction widens all 16-bit half-precision elements in the one, two, or four first source vectors and the indexed element of the second source vector to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the ZA double-vector groups.

The half-precision elements within the second source vector are specified using a 3-bit immediate index which selects the same element position within each 128-bit vector segment.

The lowest of the two consecutive vector numbers forming the double-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: [One ZA double-vector](#) , [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

### One ZA double-vector

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	0	Zm			i3h	Rv	1	i3l	Zn			0	0	off3			S				

**FMLAL** ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = FALSE;
constant integer nreg = 1;
```

### Two ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm			0	Rv	1	i3h	Zn			0	0	0	i3l	off2			S		

**FMLAL** ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = FALSE;
constant integer nreg = 2;
```

### Four ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm			1	Rv	1	i3h	Zn			0	0	0	0	i3l	off2			S	

```
FMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = FALSE;
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
<offs2>	For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV 32;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 1
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer segmentbase = e - (e MOD eltspersegment);
      integer s = 2 * segmentbase + index;
      bits(16) element1 = Elem[operand1, 2 * e + i, 16];
      bits(16) element2 = Elem[operand2, s, 16];
      bits(32) element3 = Elem[operand3, e, 32];
      if sub_op then element1 = FPNeg(element1);
      Elem[result, e, 32] = FPMulAddH\_ZA(element3, element1, element2, FPCR[]);
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMLAL (multiple and single vector)

Multi-vector floating-point multiply-add long by vector

This half-precision floating-point multiply-add long instruction widens all 16-bit half-precision elements in the one, two, or four first source vectors and the second source vector to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: [One ZA double-vector](#) , [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

### One ZA double-vector

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			0	Rv		0	1	1	Zn			0	0	off3			S		

FMLAL ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
boolean sub_op = FALSE;
constant integer nreg = 1;
```

### Two ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			0	Rv		0	1	0	Zn			0	0	0	off2			S	

FMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
boolean sub_op = FALSE;
constant integer nreg = 2;
```

### Four ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm			0	Rv		0	1	0	Zn			0	0	0	off2			S	

```
FMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
boolean sub_op = FALSE;
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
<offs2>	For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[(n+r) MOD 32, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 1
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      bits(16) element1 = Elem[operand1, 2 * e + i, 16];
      bits(16) element2 = Elem[operand2, 2 * e + i, 16];
      bits(32) element3 = Elem[operand3, e, 32];
      if sub_op then element1 = FPNeg(element1);
      Elem[result, e, 32] = FPMulAddH_ZA(element3, element1, element2, FPCR[]);
      ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# FMLAL (multiple vectors)

Multi-vector floating-point multiply-add long

This half-precision floating-point multiply-add long instruction widens all 16-bit half-precision elements in the two or four first and second source vectors to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

## Two ZA double-vectors (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	0	Rv		0	1	0	Zn			0	0	0	0	off2		S	

FMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off2:'0');
boolean sub_op = FALSE;
constant integer nreg = 2;
```

## Four ZA double-vectors (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	0	0	0	0	1	1	0	1	Zm			0	1	0	Rv		0	1	0	Zn			0	0	0	0	0	off2		S	

FMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off2:'0');
boolean sub_op = FALSE;
constant integer nreg = 4;
```

## Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs1> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
- <offs2> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
- <Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.



For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  
For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            bits(16) element1 = Elem[operand1, 2 * e + i, 16];
            bits(16) element2 = Elem[operand2, 2 * e + i, 16];
            bits(32) element3 = Elem[operand3, e, 32];
            if sub_op then element1 = FPNeg(element1);
            Elem[result, e, 32] = FPMulAddH_ZA(element3, element1, element2, FPCR[]);
            ZAvector[vec + i, VL] = result;
        vec = vec + vstride;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMLS (multiple and indexed vector)

Multi-vector floating-point fused multiply-subtract by indexed element

Multiply the indexed element of the second source vector by the corresponding floating-point elements of the two or four first source vectors and destructively subtract without intermediate rounding from the corresponding elements of the ZA single-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 2 bits depending on the size of the element. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.F64F64 indicates whether the double-precision variant is implemented, and

ID\_AA64SMFR0\_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 6 classes: [Two ZA single-vectors of half precision elements](#) , [Two ZA single-vectors of single precision elements](#) , [Two ZA single-vectors of double precision elements](#) , [Four ZA single-vectors of half precision elements](#) , [Four ZA single-vectors of single precision elements](#) and [Four ZA single-vectors of double precision elements](#)

### Two ZA single-vectors of half precision elements

(FEAT\_SME\_F16F16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				0	Rv		1	i3h		Zn				0	1	i3l		off3	
S																															

**FMLS** ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i3h:i3l);
boolean sub_op = TRUE;
constant integer nreg = 2;
```

### Two ZA single-vectors of single precision elements

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				0	Rv		0	i2		Zn				0	1	0	off3		
S																															

**FMLS** ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.S-<Zn2>.S }, <Zm>.S[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
boolean sub_op = TRUE;
constant integer nreg = 2;
```

## Two ZA single-vectors of double precision elements

(FEAT\_SME\_F64F64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm			0	Rv		0	0	i1	Zn			0	1	0	off3				
S																															

```
FMLS ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.D-<Zn4>.D }, <Zm>.D[<index>]
```

```
if !(HaveSME2() && HaveSMEF64F64()) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 64;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i1);
boolean sub_op = TRUE;
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs>	Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
<Zn1>	For the two ZA single-vectors of double precision elements, two ZA single-vectors of half precision elements and two ZA single-vectors of single precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA single-vectors of double precision elements, four ZA single-vectors of half precision elements and four ZA single-vectors of single precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	For the four ZA single-vectors of half precision elements and two ZA single-vectors of half precision elements variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.  For the four ZA single-vectors of single precision elements and two ZA single-vectors of single precision elements variant: is the element index, in the range 0 to 3, encoded in the "i2" field.  For the four ZA single-vectors of double precision elements and two ZA single-vectors of double precision elements variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltsperssegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        integer segmentbase = e - (e MOD eltsperssegment);
        integer s = segmentbase + index;
        bits(esize) element2 = Elem[operand2, s, esize];
        bits(esize) element3 = Elem[operand3, e, esize];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAdd_ZA(element3, element1, element2, FPCR[]);
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```



## FMLS (multiple and single vector)

Multi-vector floating-point fused multiply-subtract by vector

Multiply the corresponding floating-point elements of the two or four first source vector with corresponding elements of the second source vector and destructively subtract without intermediate rounding from the corresponding elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.F64F64 indicates whether the double-precision variant is implemented, and

ID\_AA64SMFR0\_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 4 classes: [Two ZA single-vectors](#) , [Two ZA single-vectors of half precision elements](#) , [Four ZA single-vectors](#) and [Four ZA single-vectors of half precision elements](#)

### Two ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm		0		Rv		1	1	0	Zn			0		1	off3				
																S															

**FMLS** ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
boolean sub_op = TRUE;
constant integer nreg = 2;
```

### Two ZA single-vectors of half precision elements

(FEAT\_SME\_F16F16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm		0		Rv		1			1	1	Zn			0		1	off3		
SZ																S															

**FMLS** ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
boolean sub_op = TRUE;
constant integer nreg = 2;
```

### Four ZA single-vectors

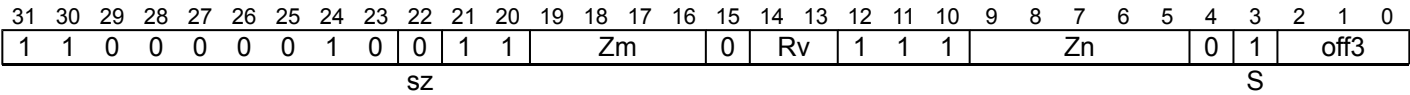
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm		0		Rv		1	1	0	Zn			0		1	off3				
																S															

FMLS ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
boolean sub_op = TRUE;
constant integer nreg = 4;
```

Four ZA single-vectors of half precision elements  
(FEAT\_SME\_F16F16)



FMLS ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
boolean sub_op = TRUE;
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
  bits(VL) operand1 = Z[(n+r) MOD 32, VL];
  bits(VL) operand2 = Z[m, VL];
  bits(VL) operand3 = ZAvector[vec, VL];
  for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    bits(esize) element3 = Elem[operand3, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd\_ZA(element3, element1, element2, FPCR[]);
  ZAvector[vec, VL] = result;
  vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## FMLS (multiple vectors)

Multi-vector floating-point fused multiply-subtract

Multiply the corresponding floating-point elements of the two or four first and second source vectors and destructively subtract without intermediate rounding from the corresponding elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.F64F64 indicates whether the double-precision variant is implemented, and

ID\_AA64SMFR0\_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 4 classes: [Two ZA single-vectors](#) , [Two ZA single-vectors of half precision elements](#) , [Four ZA single-vectors](#) and [Four ZA single-vectors of half precision elements](#)

### Two ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1		Zm		0	0		Rv	1	1	0		Zn		0	0	1		off3			
																S															

**FMLS** ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
boolean sub_op = TRUE;
constant integer nreg = 2;
```

### Two ZA single-vectors of half precision elements

(FEAT\_SME\_F16F16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm			0		0	Rv		1	0	0	Zn			0		1	1	off3		
SZ										S																					

**FMLS** ZA.H[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
boolean sub_op = TRUE;
constant integer nreg = 2;
```

### Four ZA single-vectors

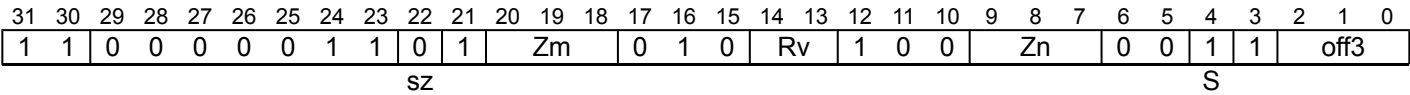
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1			Zm		0	1	0		Rv	1	1	0			Zn		0	0	0	1	off3
																S															

FMLS ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
boolean sub_op = TRUE;
constant integer nreg = 4;
```

Four ZA single-vectors of half precision elements  
(FEAT\_SME\_F16F16)



FMLS ZA.H[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
boolean sub_op = TRUE;
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors and two ZA single-vectors of half precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA single-vectors and four ZA single-vectors of half precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA single-vectors and two ZA single-vectors of half precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  
For the four ZA single-vectors and four ZA single-vectors of half precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```
CheckStreamingsVEAndZAEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m+r, VL];
  bits(VL) operand3 = ZAvector[vec, VL];
  for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    bits(esize) element3 = Elem[operand3, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd\_ZA(element3, element1, element2, FPCR[]);
  ZAvector[vec, VL] = result;
  vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMLSL (multiple and indexed vector)

Multi-vector floating-point multiply-subtract long by indexed element

This half-precision floating-point multiply-subtract long instruction widens all 16-bit half-precision elements in the one, two, or four first source vectors and the indexed element of the second source vector to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the ZA double-vector groups.

The half-precision elements within the second source vector are specified using a 3-bit immediate index which selects the same element position within each 128-bit vector segment.

The lowest of the two consecutive vector numbers forming the double-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: [One ZA double-vector](#) , [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

One ZA double-vector

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	0	Zm			i3h	Rv	1	i3l	Zn			0	1	off3			S				

FMLSL ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = TRUE;
constant integer nreg = 1;
```

Two ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm			0	Rv	1	i3h	Zn			0	0	1	i3l	off2			S		

FMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = TRUE;
constant integer nreg = 2;
```

Four ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	0	0	1	Zm			1	Rv	1	i3h	Zn			0	0	0	1	i3l	off2			S		

```
FMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = TRUE;
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
<offs2>	For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV 32;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 1
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer segmentbase = e - (e MOD eltspersegment);
      integer s = 2 * segmentbase + index;
      bits(16) element1 = Elem[operand1, 2 * e + i, 16];
      bits(16) element2 = Elem[operand2, s, 16];
      bits(32) element3 = Elem[operand3, e, 32];
      if sub_op then element1 = FPNeg(element1);
      Elem[result, e, 32] = FPMulAddH\_ZA(element3, element1, element2, FPCR[]);
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMLSL (multiple and single vector)

Multi-vector floating-point multiply-subtract long by vector

This half-precision floating-point multiply-subtract long instruction widens all 16-bit half-precision elements in the one, two, or four first source vectors and the second source vector to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors. The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME ZA-targeting floating-point behaviors. This instruction is unpredicated. It has encodings from 3 classes: [One ZA double-vector](#) , [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

One ZA double-vector  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			0	Rv		0	1	1	Zn			0	1	off3			S		

FMLSL ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
boolean sub_op = TRUE;
constant integer nreg = 1;
```

Two ZA double-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			0	Rv		0	1	0	Zn			0	1	0	off2			S	

FMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
boolean sub_op = TRUE;
constant integer nreg = 2;
```

Four ZA double-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm			0	Rv		0	1	0	Zn			0	1	0	off2			S	

```
FMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
boolean sub_op = TRUE;
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
<offs2>	For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[(n+r) MOD 32, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 1
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      bits(16) element1 = Elem[operand1, 2 * e + i, 16];
      bits(16) element2 = Elem[operand2, 2 * e + i, 16];
      bits(32) element3 = Elem[operand3, e, 32];
      if sub_op then element1 = FPNeg(element1);
      Elem[result, e, 32] = FPMulAddH_ZA(element3, element1, element2, FPCR[]);
      ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



FMLSL (multiple vectors)

Multi-vector floating-point multiply-subtract long

This half-precision floating-point multiply-subtract long instruction widens all 16-bit half-precision elements in the two or four first and second source vectors to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

Two ZA double-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm				0	0	Rv		0	1	0	Zn				0	0	1	0	off2	
S																															

FMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off2:'0');
boolean sub_op = TRUE;
constant integer nreg = 2;
```

Four ZA double-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	1	0	Rv		0	1	0	Zn			0	0	0	1	0	off2	
S																															

FMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off2:'0');
boolean sub_op = TRUE;
constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs1> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
- <offs2> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
- <Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  
For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m+r, VL];
  for i = 0 to 1
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      bits(16) element1 = Elem[operand1, 2 * e + i, 16];
      bits(16) element2 = Elem[operand2, 2 * e + i, 16];
      bits(32) element3 = Elem[operand3, e, 32];
      if sub_op then element1 = FPNeg(element1);
      Elem[result, e, 32] = FPMulAddH_ZA(element3, element1, element2, FPCR[]);
      ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FMOPA (non-widening)

Floating-point outer product and accumulate

The half-precision variant works with a 16-bit element ZA tile.

The single-precision variant works with a 32-bit element ZA tile.

The double-precision variant works with a 64-bit element ZA tile.

These instructions generate an outer product of the first source vector and the second source vector. In case of the half-precision variant, the first source is  $SVL_H \times 1$  vector and the second source is  $1 \times SVL_H$  vector. In case of the single-precision variant, the first source is  $SVL_S \times 1$  vector and the second source is  $1 \times SVL_S$  vector. In case of the double-precision variant, the first source is  $SVL_D \times 1$  vector and the second source is  $1 \times SVL_D$  vector.

Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is Inactive the corresponding destination tile element remains unmodified.

The resulting outer product,  $SVL_H \times SVL_H$  in case of half-precision variant,  $SVL_S \times SVL_S$  in case of single-precision variant or  $SVL_D \times SVL_D$  in case of double-precision variant, is then destructively added to the destination tile. This is equivalent to performing a single multiply-accumulate to each of the destination tile elements.

This instruction follows SME ZA-targeting floating-point behaviors.

ID\_AA64SMFR0\_EL1.F64F64 indicates whether the double-precision variant is implemented, and

ID\_AA64SMFR0\_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 3 classes: [Half-precision](#) , [Single-precision](#) and [Double-precision](#)

Half-precision  
(FEAT\_SME\_F16F16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	0	0	0	0	0	0	1	1	0	0	Zm				Pm			Pn			Zn				0	1	0	0	ZAda						
																															S				

FMOPA <ZAda>.H, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
constant integer esize = 16;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
```

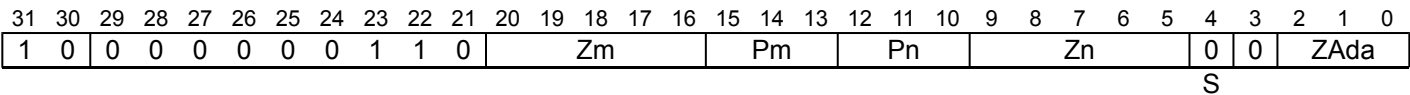
Single-precision  
(FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	0	0	0	0	0	0	0	1	0	0	Zm				Pm			Pn			Zn				0	0	0	ZAda							
																															S				

FMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S, <Zm>.S

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
```

Double-precision  
(FEAT\_SME\_F64F64)



FMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D, <Zm>.D

```
if !HaveSMEF64F64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
```

Assembler Symbols

- <ZAda> For the half-precision variant: is the name of the ZA tile ZA0-ZA1, encoded in the "ZAda" field.  
For the single-precision variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.  
For the double-precision variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) element1 = Elem[operand1, row, esize];
    bits(esize) element2 = Elem[operand2, col, esize];
    bits(esize) element3 = Elem[operand3, row*dim+col, esize];

    if (ActivePredicateElement(mask1, row, esize) &&
        ActivePredicateElement(mask2, col, esize)) then
      if sub_op then element1 = FPNeg(element1);
      Elem[result, row*dim+col, esize] = FPMulAdd_ZA(element3, element1, element2, FPCR[]);
    else
      Elem[result, row*dim+col, esize] = element3;

ZAtile[da, esize, dim*dim*esize] = result;
```

FMOPA (widening)

Half-precision floating-point sum of outer products and accumulate

The half-precision floating-point sum of outer products and accumulate instruction works with a 32-bit element ZA tile. This instruction widens the  $SVL_S \times 2$  sub-matrix of half-precision floating-point values held in the first source vector to single-precision floating-point values and multiplies it by the widened  $2 \times SVL_S$  sub-matrix of half-precision floating-point values in the second source vector to single-precision floating-point values.

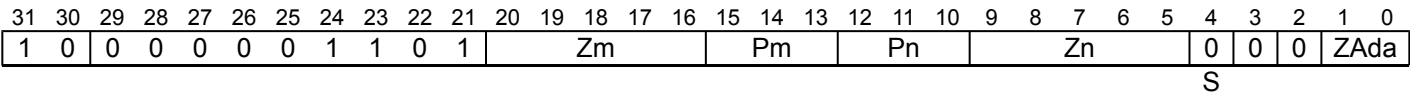
Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting  $SVL_S \times SVL_S$  single-precision floating-point sum of outer products is then destructively added to the single-precision floating-point destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a  $SVL_S \times 2$  sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a  $2 \times SVL_S$  sub-matrix.

This instruction follows SME ZA-targeting floating-point behaviors.

SME
(FEAT\_SME)



FMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSME() then UNDEFINED;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
```

Assembler Symbols

- <ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV 32;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
bits(dim*dim*32) result;

for row = 0 to dim-1
  for col = 0 to dim-1
    // determine row/col predicates
    boolean prow_0 = (ActivePredicateElement(mask1, 2*row + 0, 16));
    boolean prow_1 = (ActivePredicateElement(mask1, 2*row + 1, 16));
    boolean pcol_0 = (ActivePredicateElement(mask2, 2*col + 0, 16));
    boolean pcol_1 = (ActivePredicateElement(mask2, 2*col + 1, 16));

    bits(32) sum = Elem[operand3, row*dim+col, 32];
    if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
      bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0', 16));
      bits(16) erow_1 = (if prow_1 then Elem[operand1, 2*row + 1, 16] else FPZero('0', 16));
      bits(16) ecol_0 = (if pcol_0 then Elem[operand2, 2*col + 0, 16] else FPZero('0', 16));
      bits(16) ecol_1 = (if pcol_1 then Elem[operand2, 2*col + 1, 16] else FPZero('0', 16));
      if sub_op then
        if prow_0 then erow_0 = FPNeg(erow_0);
        if prow_1 then erow_1 = FPNeg(erow_1);
      sum = FPDotAdd\_ZA(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR[]);

    Elem[result, row*dim+col, 32] = sum;

ZAtile[da, 32, dim*dim*32] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## FMOPS (non-widening)

Floating-point outer product and subtract

The half-precision variant works with a 16-bit element ZA tile.

The single-precision variant works with a 32-bit element ZA tile.

The double-precision variant works with a 64-bit element ZA tile.

These instructions generate an outer product of the first source vector and the second source vector. In case of the half-precision variant, the first source is  $SVL_H \times 1$  vector and the second source is  $1 \times SVL_H$  vector. In case of the single-precision variant, the first source is  $SVL_S \times 1$  vector and the second source is  $1 \times SVL_S$  vector. In case of the double-precision variant, the first source is  $SVL_D \times 1$  vector and the second source is  $1 \times SVL_D$  vector.

Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is Inactive the corresponding destination tile element remains unmodified.

The resulting outer product,  $SVL_H \times SVL_H$  in case of half-precision variant,  $SVL_S \times SVL_S$  in case of single-precision variant or  $SVL_D \times SVL_D$  in case of double-precision variant, is then destructively subtracted from the destination tile. This is equivalent to performing a single multiply-subtract from each of the destination tile elements.

This instruction follows SME ZA-targeting floating-point behaviors.

ID\_AA64SMFR0\_EL1.F64F64 indicates whether the double-precision variant is implemented, and

ID\_AA64SMFR0\_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 3 classes: [Half-precision](#) , [Single-precision](#) and [Double-precision](#)

### Half-precision

(FEAT\_SME\_F16F16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	1	0	0	Zm				Pm			Pn			Zn				1	1	0	0	ZAda		
S																															

FMOPS <ZAda>.H, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
constant integer esize = 16;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
```

### Single-precision

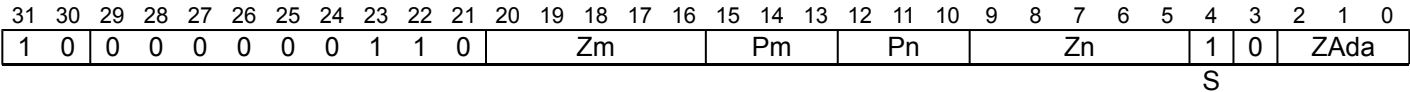
(FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	0	0	0	0	0	0	0	1	0	0	Zm				Pm			Pn			Zn				1	0	0	ZAda							
																															S				

FMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S, <Zm>.S

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
```

Double-precision  
(FEAT\_SME\_F64F64)



FMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D, <Zm>.D

```
if !HaveSMEF64F64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
```

Assembler Symbols

- <ZAda> For the half-precision variant: is the name of the ZA tile ZA0-ZA1, encoded in the "ZAda" field.  
For the single-precision variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.  
For the double-precision variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) element1 = Elem[operand1, row, esize];
    bits(esize) element2 = Elem[operand2, col, esize];
    bits(esize) element3 = Elem[operand3, row*dim+col, esize];

    if (ActivePredicateElement(mask1, row, esize) &&
        ActivePredicateElement(mask2, col, esize)) then
      if sub_op then element1 = FPNeg(element1);
      Elem[result, row*dim+col, esize] = FPMulAdd_ZA(element3, element1, element2, FPCR[]);
    else
      Elem[result, row*dim+col, esize] = element3;

ZAtile[da, esize, dim*dim*esize] = result;
```



FMOPS (widening)

Half-precision floating-point sum of outer products and subtract

The half-precision floating-point sum of outer products and subtract instruction works with a 32-bit element ZA tile. This instruction widens the  $SVL_S \times 2$  sub-matrix of half-precision floating-point values held in the first source vector to single-precision floating-point values and multiplies it by the widened  $2 \times SVL_S$  sub-matrix of half-precision floating-point values in the second source vector to single-precision floating-point values.

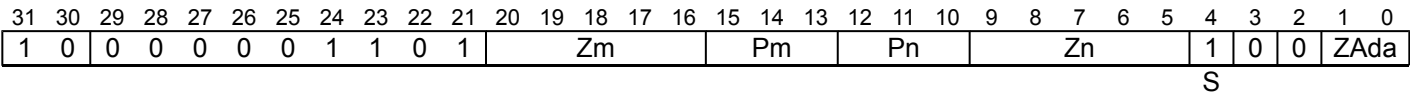
Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting  $SVL_S \times SVL_S$  single-precision floating-point sum of outer products is then destructively subtracted from the single-precision floating-point destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a  $SVL_S \times 2$  sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a  $2 \times SVL_S$  sub-matrix.

This instruction follows SME ZA-targeting floating-point behaviors.

SME
(FEAT\_SME)



FMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSME() then UNDEFINED;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
```

Assembler Symbols

- <ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV 32;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
bits(dim*dim*32) result;

for row = 0 to dim-1
  for col = 0 to dim-1
    // determine row/col predicates
    boolean prow_0 = (ActivePredicateElement(mask1, 2*row + 0, 16));
    boolean prow_1 = (ActivePredicateElement(mask1, 2*row + 1, 16));
    boolean pcol_0 = (ActivePredicateElement(mask2, 2*col + 0, 16));
    boolean pcol_1 = (ActivePredicateElement(mask2, 2*col + 1, 16));

    bits(32) sum = Elem[operand3, row*dim+col, 32];
    if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
      bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0', 16));
      bits(16) erow_1 = (if prow_1 then Elem[operand1, 2*row + 1, 16] else FPZero('0', 16));
      bits(16) ecol_0 = (if pcol_0 then Elem[operand2, 2*col + 0, 16] else FPZero('0', 16));
      bits(16) ecol_1 = (if pcol_1 then Elem[operand2, 2*col + 1, 16] else FPZero('0', 16));
      if sub_op then
        if prow_0 then erow_0 = FPNeg(erow_0);
        if prow_1 then erow_1 = FPNeg(erow_1);
      sum = FPDotAdd\_ZA(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR[]);

    Elem[result, row*dim+col, 32] = sum;

ZAtile[da, 32, dim*dim*32] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# FRINTA

Multi-vector floating-point round to integral value, to nearest with ties away from zero

Round to the nearest integral floating-point value, with ties rounding away from zero, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

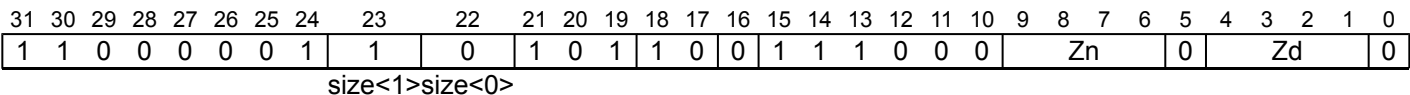
This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

## Two registers

(FEAT\_SME2)

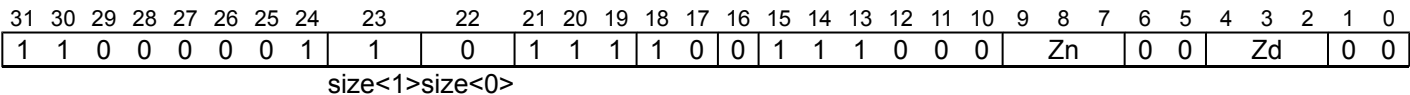


FRINTA { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd:'0');
constant integer nreg = 2;
boolean exact = FALSE;
FPRounding rounding = FPRounding_TIEAWAY;
```

## Four registers

(FEAT\_SME2)



FRINTA { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'00');
integer d = UInt(Zd:'00');
constant integer nreg = 4;
boolean exact = FALSE;
FPRounding rounding = FPRounding_TIEAWAY;
```

## Assembler Symbols

- <Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.  
For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(32) element = Elem[operand, e, 32];
        Elem[results[r], e, 32] = FPRoundInt(element, FPCR[], rounding, exact);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTM

Multi-vector floating-point round to integral value, toward minus Infinity

Round down to an integral floating-point value, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	0	1	0	1	0	1	1	1	0	0	0	Zn			0	Zd			0		
size<1>size<0>																															

```
FRINTM { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd:'0');
constant integer nreg = 2;
boolean exact = FALSE;
FPRounding rounding = FPRounding_NEGINF;
```

Four registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	0	1	1	1	0	1	0	1	1	1	0	0	0	Zn			0		0	Zd		0		0
size<1>size<0>																																

```
FRINTM { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'00');
integer d = UInt(Zd:'00');
constant integer nreg = 4;
boolean exact = FALSE;
FPRounding rounding = FPRounding_NEGINF;
```

Assembler Symbols

- <Zd1>

For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <Zd4>

Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2>

Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn1>

For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(32) element = Elem[operand, e, 32];
        Elem[results[r], e, 32] = FPRoundInt(element, FPCR[], rounding, exact);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FRINTN

Multi-vector floating-point round to integral value, to nearest with ties to even

Round to the nearest integral floating-point value, with ties rounding to an even value, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

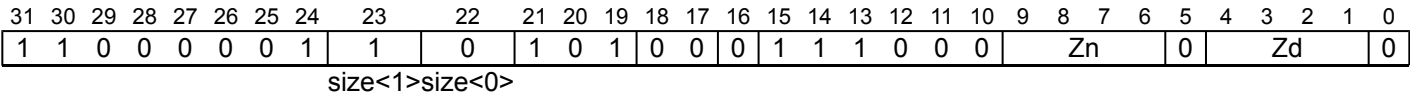
This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers

(FEAT\_SME2)

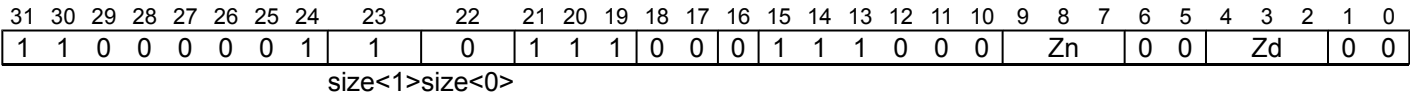


```
FRINTN { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd:'0');
constant integer nreg = 2;
boolean exact = FALSE;
FPRounding rounding = FPRounding_TIEEVEN;
```

Four registers

(FEAT\_SME2)



```
FRINTN { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'00');
integer d = UInt(Zd:'00');
constant integer nreg = 4;
boolean exact = FALSE;
FPRounding rounding = FPRounding_TIEEVEN;
```

Assembler Symbols

- <Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.  
For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(32) element = Elem[operand, e, 32];
        Elem[results[r], e, 32] = FPRoundInt(element, FPCR[], rounding, exact);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



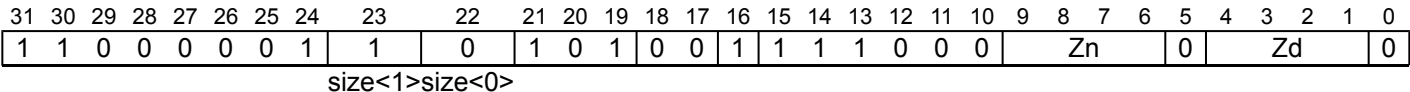
FRINTP

Multi-vector floating-point round to integral value, toward plus Infinity

Round up to an integral floating-point value, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.  
This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.  
It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

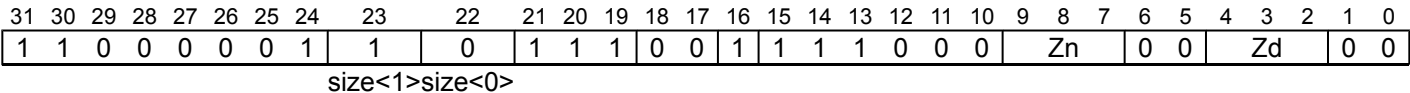
Two registers  
(FEAT\_SME2)



```
FRINTP { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd:'0');
constant integer nreg = 2;
boolean exact = FALSE;
FPRounding rounding = FPRounding_POSINF;
```

Four registers  
(FEAT\_SME2)



```
FRINTP { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'00');
integer d = UInt(Zd:'00');
constant integer nreg = 4;
boolean exact = FALSE;
FPRounding rounding = FPRounding_POSINF;
```

Assembler Symbols

- <Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.  
For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(32) element = Elem[operand, e, 32];
        Elem[results[r], e, 32] = FPRoundInt(element, FPCR[], rounding, exact);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FSUB

Floating-point subtract multi-vector from ZA array vector accumulators

Destructively subtract all elements of the two or four source vectors from the corresponding elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.F64F64 indicates whether the double-precision variant is implemented, and

ID\_AA64SMFR0\_EL1.F16F16 indicates whether the half-precision variant is implemented.

It has encodings from 4 classes: [Two ZA single-vectors](#) , [Two ZA single-vectors of half precision elements](#) , [Four ZA single-vectors](#) and [Four ZA single-vectors of half precision elements](#)

Two ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	0	0	Rv	1	1	1	Zm			0	0	1	off3				
S																															

FSUB ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Two ZA single-vectors of half precision elements

(FEAT\_SME\_F16F16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	0	0	1	0	0	0	Rv	1	1	1	Zm			0	0	1	off3				
SZ																				S											

FSUB ZA.H[<Wv>, <offs>{, VGx2}], { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors

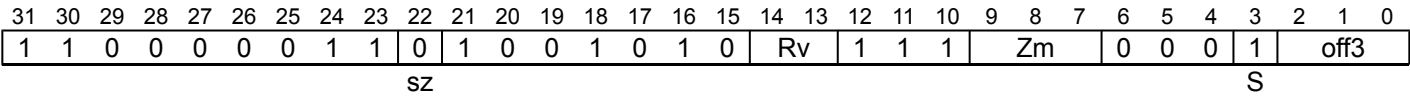
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	1	0	Rv	1	1	1	Zm			0	0	0	1	off3			
S																															

FSUB ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Four ZA single-vectors of half precision elements  
(FEAT\_SME\_F16F16)



FSUB ZA.H[<Wv>, <offs>{, VGx4}], { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() || !IsFeatureImplemented(FEAT_SME_F16F16) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 16;
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zm1> For the two ZA single-vectors and two ZA single-vectors of half precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  
For the four ZA single-vectors and four ZA single-vectors of half precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```
CheckStreamingSVEAndZAAEnabled\(\);  
constant integer VL = CurrentVL;  
constant integer elements = VL DIV esize;  
integer vectors = VL DIV 8;  
integer vstride = vectors DIV nreg;  
bits(32) vbase = X[v, 32];  
integer vec = (UInt(vbase) + offset) MOD vstride;  
bits(VL) result;  
  
for r = 0 to nreg-1  
  bits(VL) operand1 = ZAvector[vec, VL];  
  bits(VL) operand2 = Z[m+r, VL];  
  for e = 0 to elements-1  
    bits(esize) element1 = Elem[operand1, e, esize];  
    bits(esize) element2 = Elem[operand2, e, esize];  
    Elem[result, e, esize] = FPSub\_ZA(element1, element2, FPCR[]);  
  ZAvector[vec, VL] = result;  
  vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FVDDOT

Multi-vector half-precision floating-point vertical dot-product by indexed element

The instruction computes the fused sum-of-products of each vertical pair of half-precision floating-point values in the corresponding elements of the two first source vectors with the pair of half-precision floating-point values in the indexed 32-bit group of the corresponding 128-bit segment of the second source vector, without intermediate rounding. The single-precision sum-of-products results are destructively added to the corresponding single-precision elements of the two ZA single-vector groups.

The half-precision floating-point pairs within the second source vector are specified using an immediate index which selects the same pair position within each 128-bit vector segment. The element index range is from 0 to 3.

The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

The VECTOR GROUP symbol VGx2 indicates that the ZA operand consists of two ZA single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

SME2  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	0	1	Zm			0	Rv		0	i2		Zn			0	0	1	off3				

FVDDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> Is the immediate index of a group of two 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

## Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV 2;
integer eltspersegment = 128 DIV 32;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to 1
    bits(VL) operand1a = Z[n, VL];
    bits(VL) operand1b = Z[n+1, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        bits(16) elt1_a = Elem[operand1a, 2 * e + r, 16];
        bits(16) elt1_b = Elem[operand1b, 2 * e + r, 16];
        bits(16) elt2_a = Elem[operand2, 2 * s + 0, 16];
        bits(16) elt2_b = Elem[operand2, 2 * s + 1, 16];
        bits(32) sum = Elem[operand3, e, 32];
        sum = FPDotAdd ZA(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
        Elem[result, e, 32] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1B (scalar plus immediate, strided registers)

Contiguous load of bytes to multiple strided vectors (immediate index)

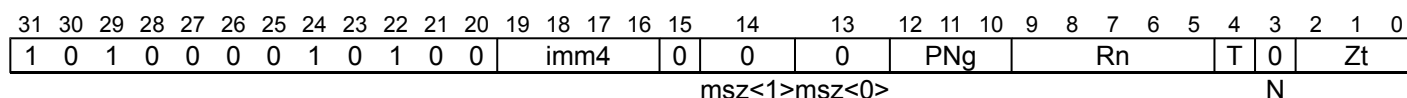
Contiguous load of unsigned bytes to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

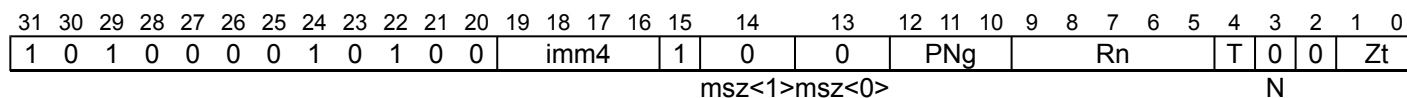


LD1B { <Zt1>.B, <Zt2>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 8;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SME2)



LD1B { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 8;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".



<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.  For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1B (scalar plus scalar, strided registers)

Contiguous load of bytes to multiple strided vectors (scalar index)

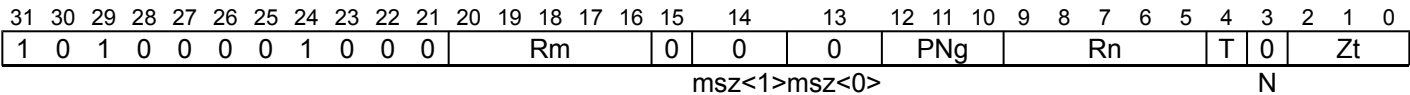
Contiguous load of unsigned bytes to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

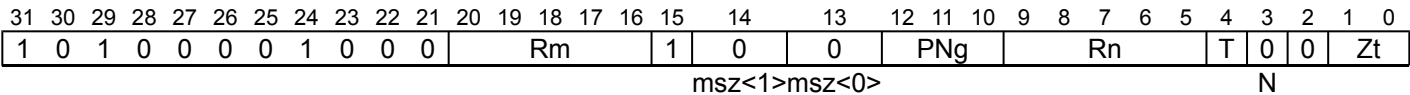


LD1B { <Zt1>.B, <Zt2>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 8;
```

### Four registers

(FEAT\_SME2)



LD1B { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 8;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

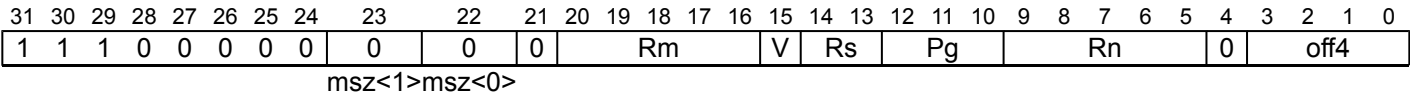
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1B (scalar plus scalar, tile slice)

Contiguous load of bytes to 8-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 8-bit elements in a vector. The immediate offset is in the range 0 to 15. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME
(FEAT\_SME)



LD1B { ZA0<HV>.B[<Ws>, <offs>] }, <Pg>/Z, [<Xn|SP>{, <Xm>}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = 0;
integer offset = UInt(off4);
constant integer esize = 8;
boolean vertical = V == '1';
```

Assembler Symbols

- <HV> Is the horizontal or vertical slice indicator, encoded in "V":
Table:
+---+
| V | <HV> |
+---+
| 0 | H |
| 1 | V |
+---+
- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs> Is the slice index offset, in the range 0 to 15, encoded in the "off4" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) moffs = X[m, 64];
bits(32) index = X[s, 32];
integer slice = (UInt(index) + offset) MOD dim;
bits(VL) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
        ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

for e = 0 to dim - 1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);
    moffs = moffs + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1D (scalar plus immediate, strided registers)

Contiguous load of doublewords to multiple strided vectors (immediate index)

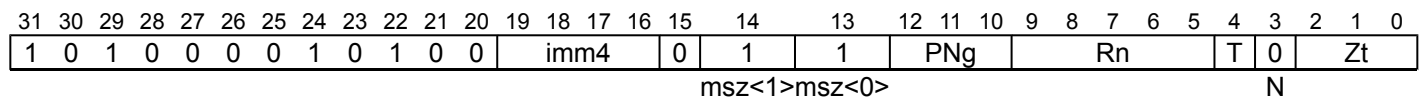
Contiguous load of unsigned doublewords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

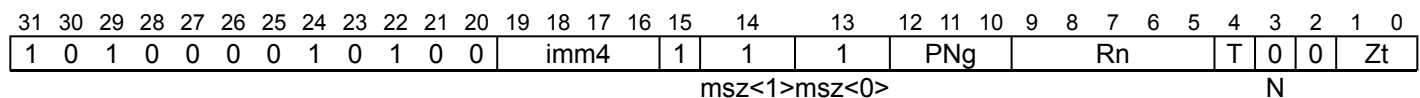


LD1D { <Zt1>.D, <Zt2>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SME2)



LD1D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.  For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1D (scalar plus scalar, strided registers)

Contiguous load of doublewords to multiple strided vectors (scalar index)

Contiguous load of unsigned doublewords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	0	1	0	0	0	0	1	0	0	0	Rm				0	1	1	PNg			Rn				T	0	Zt								
																msz<1>msz<0>										N									

LD1D { <Zt1>.D, <Zt2>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 64;
```

### Four registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	0	1	0	0	0	0	1	0	0	0	Rm				1	1	1	PNg			Rn				T	0	0	Zt								
																msz<1>msz<0>										N										

LD1D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 64;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".



<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

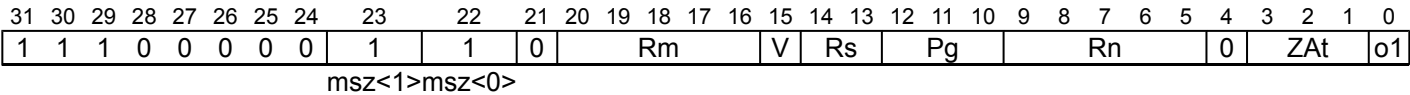
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1D (scalar plus scalar, tile slice)

Contiguous load of doublewords to 64-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 64-bit elements in a vector. The immediate offset is in the range 0 to 1. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is multiplied by 8 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME
(FEAT\_SME)



LD1D { <ZAt><HV>.D[<Ws>, <offs>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #3}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer offset = UInt(o1);
constant integer esize = 64;
boolean vertical = V == '1';
```

Assembler Symbols

- <ZAt> Is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAt" field.
- <HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V
- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs> Is the slice index offset, in the range 0 to 1, encoded in the "o1" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) moffs = X[m, 64];
bits(32) index = X[s, 32];
integer slice = (UInt(index) + offset) MOD dim;
bits(VL) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
        ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

for e = 0 to dim - 1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);
    moffs = moffs + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1H (scalar plus immediate, strided registers)

Contiguous load of halfwords to multiple strided vectors (immediate index)

Contiguous load of unsigned halfwords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	0	1	0	0	0	0	1	0	1	0	0	imm4				0	0	1	PNg			Rn				T	0	Zt							
																msz<1>msz<0>										N									

LD1H { <Zt1>.H, <Zt2>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 16;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	0	1	0	0	0	0	1	0	1	0	0	imm4				1	0	1	PNg			Rn				T	0	0	Zt							
																msz<1>msz<0>										N										

LD1H { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 16;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.  For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1H (scalar plus scalar, strided registers)

Contiguous load of halfwords to multiple strided vectors (scalar index)

Contiguous load of unsigned halfwords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	0	1	0	0	0	0	1	0	0	0	Rm				0	0	1	PNg			Rn				T	0	Zt								
																msz<1>msz<0>										N									

LD1H { <Zt1>.H, <Zt2>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 16;
```

### Four registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	0	1	0	0	0	0	1	0	0	0	Rm				1	0	1	PNg			Rn				T	0	0	Zt								
																msz<1>msz<0>										N										

LD1H { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 16;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

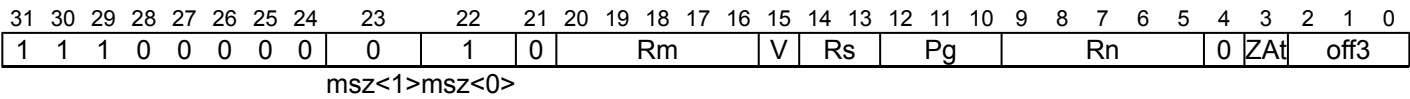
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1H (scalar plus scalar, tile slice)

Contiguous load of halfwords to 16-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 16-bit elements in a vector. The immediate offset is in the range 0 to 7. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is multiplied by 2 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME
(FEAT\_SME)



LD1H { <ZAt><HV>.H[<Ws>, <offs>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer offset = UInt(off3);
constant integer esize = 16;
boolean vertical = V == '1';
```

Assembler Symbols

- <ZAt> Is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAt" field.
- <HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V
- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs> Is the slice index offset, in the range 0 to 7, encoded in the "off3" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.



## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) moffs = X[m, 64];
bits(32) index = X[s, 32];
integer slice = (UInt(index) + offset) MOD dim;
bits(VL) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
        ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

for e = 0 to dim - 1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);
    moffs = moffs + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1Q

Contiguous load of quadwords to 128-bit element ZA tile slice

The slice number in the tile is selected by the slice index register, modulo the number of 128-bit elements in a Streaming SVE vector. The memory address is generated by scalar base and optional scalar offset which is multiplied by 16 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME
(FEAT\_SME)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 0 0 1 1 1 0 Rm V Rs Pg Rn 0 ZAt

LD1Q { <ZAt><HV>.Q[<Ws>, <offs>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #4}]

if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer offset = 0;
constant integer esize = 128;
boolean vertical = V == '1';

Assembler Symbols

- <ZAt> Is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAt" field.
- <HV> Is the horizontal or vertical slice indicator, encoded in "V":
Table with 2 columns: V, <HV>.
Row 1: 0, H
Row 2: 1, V
- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs> Is the slice index offset 0.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) moffs = X[m, 64];
bits(32) index = X[s, 32];
integer slice = (UInt(index) + offset) MOD dim;
bits(VL) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
        ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

for e = 0 to dim - 1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);
    moffs = moffs + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1W (scalar plus immediate, strided registers)

Contiguous load of words to multiple strided vectors (immediate index)

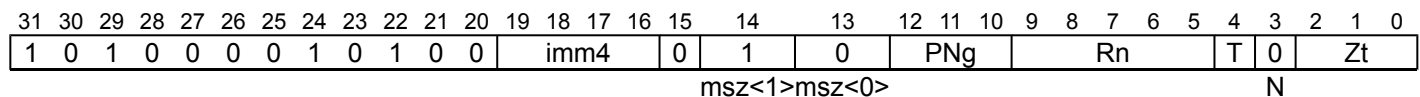
Contiguous load of unsigned words to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

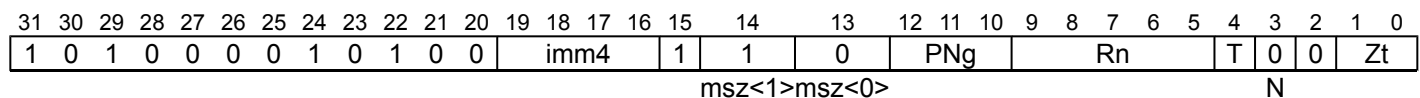


LD1W { <Zt1>.S, <Zt2>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 32;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SME2)



LD1W { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 32;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.  For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LD1W (scalar plus scalar, strided registers)

Contiguous load of words to multiple strided vectors (scalar index)

Contiguous load of unsigned words to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	0	Rm				0	1	0	PNg			Rn				T	0	Zt				
																msz<1>msz<0>												N			

LD1W { <Zt1>.S, <Zt2>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 32;
```

### Four registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	1	0	0	0	0	1	0	0	0	Rm				1	1	0	PNg			Rn				T	0	0	Zt				
																msz<1>msz<0>						N										

LD1W { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 32;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

    for r = 0 to nreg-1
        for e = 0 to elements-1
            if ActivePredicateElement(mask, r * elements + e, esize) then
                bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
                Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
            else
                Elem[values[r], e, esize] = Zeros(esize);

    for r = 0 to nreg-1
        Z[t, VL] = values[r];
        t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

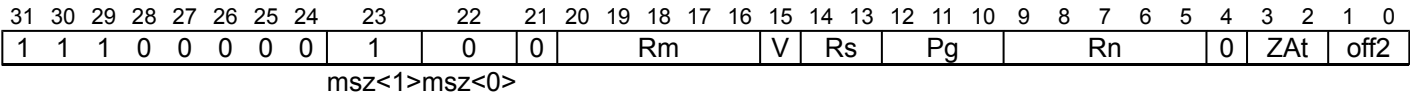
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LD1W (scalar plus scalar, tile slice)

Contiguous load of words to 32-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 32-bit elements in a vector. The immediate offset is in the range 0 to 3. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is multiplied by 4 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME
(FEAT\_SME)



LD1W { <ZAt><HV>.S[<Ws>, <offs>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer offset = UInt(off2);
constant integer esize = 32;
boolean vertical = V == '1';
```

Assembler Symbols

- <ZAt> Is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAt" field.
- <HV> Is the horizontal or vertical slice indicator, encoded in "V":
Table:
V | <HV>
0 | H
1 | V
- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs> Is the slice index offset, in the range 0 to 3, encoded in the "off2" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.



## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) moffs = X[m, 64];
bits(32) index = X[s, 32];
integer slice = (UInt(index) + offset) MOD dim;
bits(VL) result;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
        ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

for e = 0 to dim - 1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);
    moffs = moffs + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1B (scalar plus immediate, strided registers)

Contiguous load non-temporal of bytes to multiple strided vectors (immediate index)

Contiguous load non-temporal of bytes to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

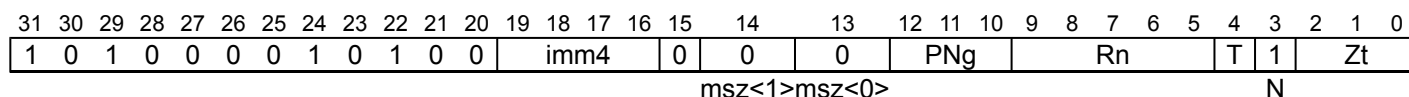
Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

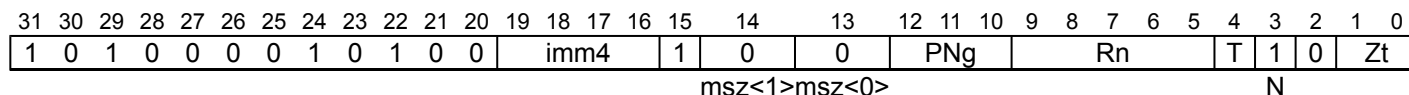


LDNT1B { <Zt1>.B, <Zt2>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 8;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SME2)



LDNT1B { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 8;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3>	Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.  For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1B (scalar plus scalar, strided registers)

Contiguous load non-temporal of bytes to multiple strided vectors (scalar index)

Contiguous load non-temporal of bytes to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

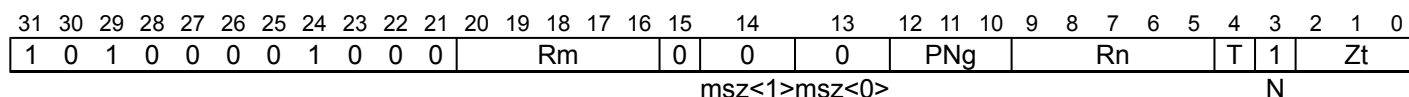
Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

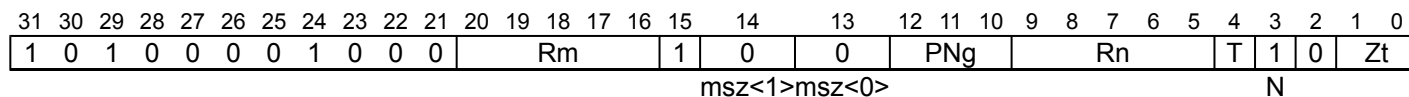


LDNT1B { <Zt1>.B, <Zt2>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 8;
```

### Four registers

(FEAT\_SME2)



LDNT1B { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 8;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3>	Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1D (scalar plus immediate, strided registers)

Contiguous load non-temporal of doublewords to multiple strided vectors (immediate index)

Contiguous load non-temporal of doublewords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

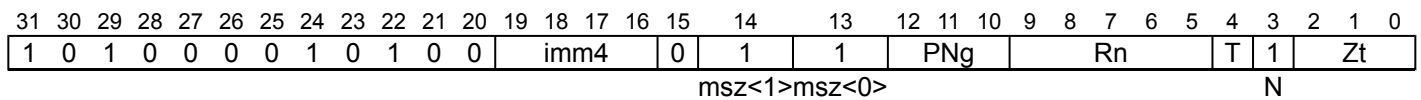
Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

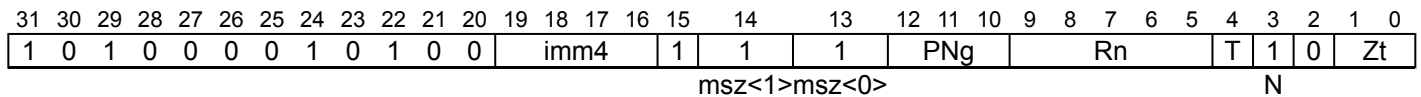


LDNT1D { <Zt1>.D, <Zt2>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SME2)



LDNT1D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3>	Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.  For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1D (scalar plus scalar, strided registers)

Contiguous load non-temporal of doublewords to multiple strided vectors (scalar index)

Contiguous load non-temporal of doublewords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

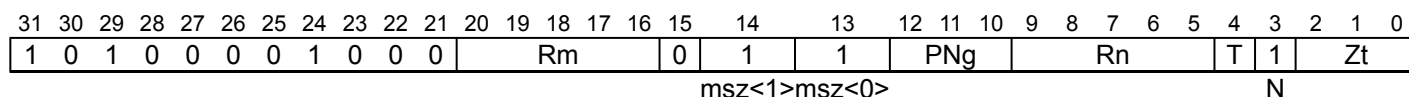
Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

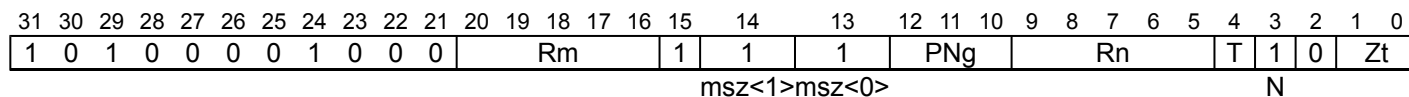


LDNT1D { <Zt1>.D, <Zt2>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 64;
```

### Four registers

(FEAT\_SME2)



LDNT1D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 64;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".



<Zt3>	Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1H (scalar plus immediate, strided registers)

Contiguous load non-temporal of halfwords to multiple strided vectors (immediate index)

Contiguous load non-temporal of halfwords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	0	0	imm4				0	0	1	PNg				Rn				T	1	Zt		
																msz<1>msz<0>								N							

LDNT1H { <Zt1>.H, <Zt2>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 16;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	0	1	0	0	0	0	1	0	1	0	0	imm4				1	0	1	PNg				Rn				T	1	0	Zt					
																msz<1>msz<0>										N									

LDNT1H { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 16;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3>	Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.  For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1H (scalar plus scalar, strided registers)

Contiguous load non-temporal of halfwords to multiple strided vectors (scalar index)

Contiguous load non-temporal of halfwords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	0	Rm				0	0	1	PNg			Rn				T	1	Zt			0	
																msz<1>msz<0>				N											

LDNT1H { <Zt1>.H, <Zt2>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 16;
```

### Four registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	0	Rm				1	0	1	PNg			Rn				T	1	0	Zt		0	
																msz<1>msz<0>				N											

LDNT1H { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 16;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3>	Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1W (scalar plus immediate, strided registers)

Contiguous load non-temporal of words to multiple strided vectors (immediate index)

Contiguous load non-temporal of words to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

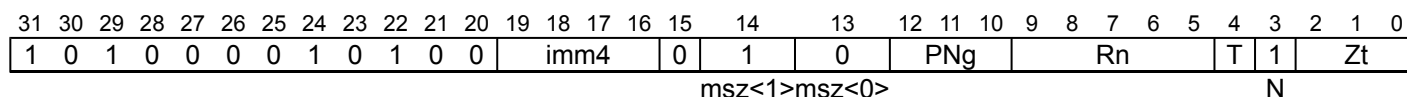
Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

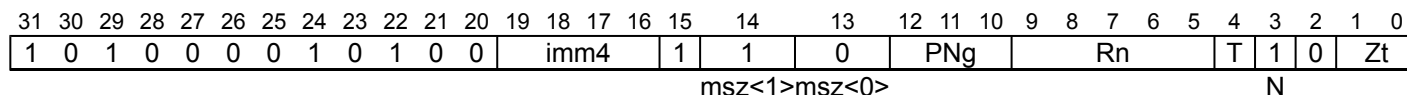


```
LDNT1W { <Zt1>.S, <Zt2>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 32;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SME2)



```
LDNT1W { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 32;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3>	Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.  For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LDNT1W (scalar plus scalar, strided registers)

Contiguous load non-temporal of words to multiple strided vectors (scalar index)

Contiguous load non-temporal of words to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	0	1	0	0	0	0	1	0	0	0	Rm				0	1	0	PNg			Rn				T	1	Zt								
																msz<1>msz<0>										N									

```
LDNT1W { <Zt1>.S, <Zt2>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 32;
```

### Four registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	0	1	0	0	0	0	1	0	0	0	Rm				1	1	0	PNg			Rn				T	1	0	Zt								
																msz<1>msz<0>										N										

```
LDNT1W { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 32;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".
- For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".
- For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".



<Zt3>	Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDR (vector)

Load ZA array vector

The ZA array vector is selected by the sum of the vector select register and immediate offset, modulo the number of bytes in a Streaming SVE vector. The immediate offset is in the range 0 to 15. The memory address is generated by a 64-bit scalar base, plus the same optional immediate offset multiplied by the current vector length in bytes. This instruction is unpredicated.

The load is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME  
(FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	Rv	0	0	0	Rn					0	off4				

```
LDR ZA[<Wv>, <offs>], [<Xn|SP>{, #<offs>, MUL VL}]
```

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer v = UInt('011':Rv);
integer offset = UInt(off4);
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W12-W15, encoded in the "Rv" field.
- <offs> Is the vector select offset and optional memory offset, in the range 0 to 15, defaulting to 0, encoded in the "off4" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
CheckSMEAndZAEnabled();
constant integer SVL = CurrentSVL;
constant integer dim = SVL DIV 8;
bits(64) base;
integer moffs = offset * dim;
bits(SVL) result;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD dim;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSME(MemOp\_LOAD, nontemporal, contiguous, tagchecked);

if HaveTME() && TSTATE.depth > 0 then
    FailTransaction(TMFailure\_ERR, FALSE);

if n == 31 then
    CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

boolean aligned = IsAligned(base + moffs, 16);

if !aligned && AlignmentEnforced() then
    AArch64.Abort(base + moffs, AlignmentFault(accdesc));

for e = 0 to dim-1
    Elem[result, e, 8] = AArch64.MemSingle[base + moffs, 1, accdesc, aligned];
    moffs = moffs + 1;

ZAvector[vec, SVL] = result;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDR (ZT0)

Load ZT0 register

Load the 64-byte ZT0 register from the memory address provided in the 64-bit scalar base register. This instruction is unpredicated.  
The load is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.  
This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME2  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	1	0	0	0	1	1	1	1	1	1	0	0	0	0	0	Rn				0	0	0	0	0	

```
LDR ZT0, [<Xn|SP>]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
```

Assembler Symbols

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckSMEEnabled();
CheckSMEZT0Enabled();
constant integer elements = 512 DIV 8;
bits(64) base;
bits(512) result;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSME(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if HaveTME() && TSTATE.depth > 0 then
    FailTransaction(TMFailure_ERR, FALSE);

if n == 31 then
    CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

boolean aligned = IsAligned(base, 16);

if !aligned && AlignmentEnforced() then
    AArch64.Abort(base, AlignmentFault(accdesc));

for e = 0 to elements-1
    Elem[result, e, 8] = AArch64.MemSingle(base + e, 1, accdesc, aligned);

ZT0[512] = result;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



LUTI2 (four registers)

Lookup table read with 2-bit indexes

Copy 8-bit, 16-bit or 32-bit elements from ZT0 to four destination vectors using packed 2-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments. This instruction is unpredicated.

It has encodings from 2 classes: [Consecutive](#) and [Strided](#)

Consecutive  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	1	1	i2	1	0	size	0	0	Zn				Zd		0	0				

LUTI2 { <Zd1>.<T>-<Zd4>.<T> }, ZT0, <Zn>[<index>]

```
if !HaveSME2() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer isize = 2;
integer n = UInt(Zn);
integer dstride = 1;
integer d = UInt(Zd:'00');
integer imm = UInt(i2);
constant integer nreg = 4;
```

Strided  
(FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	0	1	0	0	1	1	1	i2	1	0	size	0	0	Zn				D	0	0						Zd

LUTI2 { <Zd1>.<T>, <Zd2>.<T>, <Zd3>.<T>, <Zd4>.<T> }, ZT0, <Zn>[<index>]

```
if !HaveSME2p1() then UNDEFINED;
if size == '10' || size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer isize = 2;
integer n = UInt(Zn);
integer dstride = 4;
integer d = UInt(D:'00':Zd);
integer imm = UInt(i2);
constant integer nreg = 4;
```

Assembler Symbols

- <Zd1> For the consecutive variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.  
For the strided variant: is the name of the first destination scalable vector register Z0-Z3 or Z16-Z19 of a multi-vector sequence, encoded as "D:'00':Zd".

- <T> For the consecutive variant: is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	RESERVED

For the strided variant: is the size specifier, encoded in "size<0>":

size<0>	<T>
0	B
1	H

- <Zd2> Is the name of the second destination scalable vector register Z4-Z7 or Z20-Z23 of a multi-vector sequence, encoded as "D:'01':Zd".
- <Zd3> Is the name of the third destination scalable vector register Z8-Z11 or Z24-Z27 of a multi-vector sequence, encoded as "D:'10':Zd".
- <Zd4> For the consecutive variant: is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.  
For the strided variant: is the name of the fourth destination scalable vector register Z12-Z15 or Z28-Z31 of a multi-vector sequence, encoded as "D:'11':Zd".
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
- <index> Is the vector segment index, in the range 0 to 3, encoded in the "i2" field.

## Operation

```

CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer segments = esize DIV (isize * nreg);
integer segment = imm MOD segments;
bits(VL) indexes = Z[n, VL];
bits(512) table = ZT0[512];

for r = 0 to nreg-1
    integer base = (segment * nreg + r) * elements;
    bits(VL) result;
    for e = 0 to elements-1
        integer index = UInt(Elem[indexes, base+e, isize]);
        Elem[result, e, esize] = Elem[table, index, 32]<esize-1:0>;
    Z[d, VL] = result;
    d = d + dstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LUTI2 (single)

Lookup table read with 2-bit indexes

Copy 8-bit, 16-bit or 32-bit elements from ZT0 to one destination vector using packed 2-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments. This instruction is unpredicated.

SME2  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	1	1	i4			size		0		0	Zn				Zd					

LUTI2 <Zd>.<T>, ZT0, <Zn>[<index>]

```
if !HaveSME2() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer isize = 2;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer imm = UInt(i4);
constant integer nreg = 1;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	RESERVED

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<index> Is the vector segment index, in the range 0 to 15, encoded in the "i4" field.

Operation

```
CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer segments = esize DIV (isize * nreg);
integer segment = imm MOD segments;
bits(VL) indexes = Z[n, VL];
bits(512) table = ZT0[512];

for r = 0 to nreg-1
    integer base = (segment * nreg + r) * elements;
    bits(VL) result;
    for e = 0 to elements-1
        integer index = UInt(Elem[indexes, base+e, isize]);
        Elem[result, e, esize] = Elem[table, index, 32]<esize-1:0>;
    Z[d+r, VL] = result;
```



## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LUTI2 (two registers)

Lookup table read with 2-bit indexes

Copy 8-bit, 16-bit or 32-bit elements from ZT0 to two destination vectors using packed 2-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

It has encodings from 2 classes: [Consecutive](#) and [Strided](#)

Consecutive  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	1	1	i3		1		size		0	0	Zn				Zd			0		

LUTI2 { <Zd1>.<T>-<Zd2>.<T> }, ZT0, <Zn>[<index>]

```
if !HaveSME2() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer isize = 2;
integer n = UInt(Zn);
integer dstride = 1;
integer d = UInt(Zd:'0');
integer imm = UInt(i3);
constant integer nreg = 2;
```

Strided  
(FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	1	1	1	i3		1		size	0	0	Zn				D	0	Zd				

LUTI2 { <Zd1>.<T>, <Zd2>.<T> }, ZT0, <Zn>[<index>]

```
if !HaveSME2p1() then UNDEFINED;
if size == '10' || size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer isize = 2;
integer n = UInt(Zn);
integer dstride = 8;
integer d = UInt(D:'0':Zd);
integer imm = UInt(i3);
constant integer nreg = 2;
```

Assembler Symbols

- <Zd1> For the consecutive variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.  
For the strided variant: is the name of the first destination scalable vector register Z0-Z7 or Z16-Z23 of a multi-vector sequence, encoded as "D:'0':Zd".

- <T> For the consecutive variant: is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	RESERVED

For the strided variant: is the size specifier, encoded in "size<0>":

size<0>	<T>
0	B
1	H

- <Zd2> For the consecutive variant: is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.  
For the strided variant: is the name of the second destination scalable vector register Z8-Z15 or Z24-Z31 of a multi-vector sequence, encoded as "D:'1':Zd".
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
- <index> Is the vector segment index, in the range 0 to 7, encoded in the "i3" field.

## Operation

```

CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer segments = esize DIV (isize * nreg);
integer segment = imm MOD segments;
bits(VL) indexes = Z[n, VL];
bits(512) table = ZT0[512];

for r = 0 to nreg-1
    integer base = (segment * nreg + r) * elements;
    bits(VL) result;
    for e = 0 to elements-1
        integer index = UInt(Elem[indexes, base+e, isize]);
        Elem[result, e, esize] = Elem[table, index, 32]<esize-1:0>;
    Z[d, VL] = result;
    d = d + dstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## LUTI4 (four registers)

Lookup table read with 4-bit indexes

Copy 16-bit or 32-bit elements from ZT0 to four destination vectors using packed 4-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

It has encodings from 2 classes: [Consecutive](#) and [Strided](#)

### Consecutive (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	1	0	1	i1	1	0	size	0	0				Zn				Zd		0	0

LUTI4 { <Zd1>.<T>-<Zd4>.<T> }, ZT0, <Zn>[<index>]

```
if !HaveSME2() then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer isize = 4;
integer n = UInt(Zn);
integer dstride = 1;
integer d = UInt(Zd:'00');
integer imm = UInt(i1);
constant integer nreg = 4;
```

### Strided (FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	1	1	0	1	i1	1	0	size	0	0	Zn					D	0	0	Zd		

LUTI4 { <Zd1>.H, <Zd2>.H, <Zd3>.H, <Zd4>.H }, ZT0, <Zn>[<index>]

```
if !HaveSME2p1() then UNDEFINED;
if size != '01' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer isize = 4;
integer n = UInt(Zn);
integer dstride = 4;
integer d = UInt(D:'00':Zd);
integer imm = UInt(i1);
constant integer nreg = 4;
```

### Assembler Symbols

- <Zd1> For the consecutive variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.  
For the strided variant: is the name of the first destination scalable vector register Z0-Z3 or Z16-Z19 of a multi-vector sequence, encoded as "D:'00':Zd".
- <Zd2> Is the name of the second destination scalable vector register Z4-Z7 or Z20-Z23 of a multi-vector sequence, encoded as "D:'01':Zd".
- <Zd3> Is the name of the third destination scalable vector register Z8-Z11 or Z24-Z27 of a multi-vector sequence, encoded as "D:'10':Zd".
- <T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	RESERVED

- <Zd4> For the consecutive variant: is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- For the strided variant: is the name of the fourth destination scalable vector register Z12-Z15 or Z28-Z31 of a multi-vector sequence, encoded as "D:'11':Zd".
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
- <index> Is the vector segment index, in the range 0 to 1, encoded in the "i1" field.

## Operation

```

CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer segments = esize DIV (isize * nreg);
integer segment = imm MOD segments;
bits(VL) indexes = Z[n, VL];
bits(512) table = ZT0[512];

for r = 0 to nreg-1
    integer base = (segment * nreg + r) * elements;
    bits(VL) result;
    for e = 0 to elements-1
        integer index = UInt(Elem[indexes, base+e, isize]);
        Elem[result, e, esize] = Elem[table, index, 32]<esize-1:0>;
    Z[d, VL] = result;
    d = d + dstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LUTI4 (single)

Lookup table read with 4-bit indexes

Copy 8-bit, 16-bit or 32-bit elements from ZT0 to one destination vector using packed 4-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments. This instruction is unpredicated.

SME2  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	1	0	1		i3		size		0	0			Zn					Zd		

LUTI4 <Zd>.<T>, ZT0, <Zn>[<index>]

```
if !HaveSME2() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer isize = 4;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer imm = UInt(i3);
constant integer nreg = 1;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	RESERVED

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<index> Is the vector segment index, in the range 0 to 7, encoded in the "i3" field.

Operation

```
CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer segments = esize DIV (isize * nreg);
integer segment = imm MOD segments;
bits(VL) indexes = Z[n, VL];
bits(512) table = ZT0[512];

for r = 0 to nreg-1
    integer base = (segment * nreg + r) * elements;
    bits(VL) result;
    for e = 0 to elements-1
        integer index = UInt(Elem[indexes, base+e, isize]);
        Elem[result, e, esize] = Elem[table, index, 32]<esize-1:0>;
    Z[d+r, VL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LUTI4 (two registers)

Lookup table read with 4-bit indexes

Copy 8-bit, 16-bit or 32-bit elements from ZT0 to two destination vectors using packed 4-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

It has encodings from 2 classes: [Consecutive](#) and [Strided](#)

Consecutive  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	1	0	1	i2	1	size	0	0	Zn				Zd				0			

LUTI4 { <Zd1>.<T>-<Zd2>.<T> }, ZT0, <Zn>[<index>]

```
if !HaveSME2() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer isize = 4;
integer n = UInt(Zn);
integer dstride = 1;
integer d = UInt(Zd:'0');
integer imm = UInt(i2);
constant integer nreg = 2;
```

Strided  
(FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	1	1	0	1	i2	1	size	0	0	Zn				D	0	Zd					

LUTI4 { <Zd1>.<T>, <Zd2>.<T> }, ZT0, <Zn>[<index>]

```
if !HaveSME2p1() then UNDEFINED;
if size == '10' || size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer isize = 4;
integer n = UInt(Zn);
integer dstride = 8;
integer d = UInt(D:'0':Zd);
integer imm = UInt(i2);
constant integer nreg = 2;
```

Assembler Symbols

- <Zd1> For the consecutive variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.  
For the strided variant: is the name of the first destination scalable vector register Z0-Z7 or Z16-Z23 of a multi-vector sequence, encoded as "D:'0':Zd".

- <T> For the consecutive variant: is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	RESERVED



For the strided variant: is the size specifier, encoded in "size<0>":

size<0>	<T>
0	B
1	H

- <Zd2> For the consecutive variant: is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.  
For the strided variant: is the name of the second destination scalable vector register Z8-Z15 or Z24-Z31 of a multi-vector sequence, encoded as "D:'1':Zd".
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
- <index> Is the vector segment index, in the range 0 to 3, encoded in the "i2" field.

## Operation

```

CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer segments = esize DIV (isize * nreg);
integer segment = imm MOD segments;
bits(VL) indexes = Z[n, VL];
bits(512) table = ZT0[512];

for r = 0 to nreg-1
    integer base = (segment * nreg + r) * elements;
    bits(VL) result;
    for e = 0 to elements-1
        integer index = UInt(Elem[indexes, base+e, isize]);
        Elem[result, e, esize] = Elem[table, index, 32]<esize-1:0>;
    Z[d, VL] = result;
    d = d + dstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (array to vector, four registers)

Move four ZA single-vector groups to four vector registers

The instruction operates on four ZA single-vector groups. The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

The VECTOR GROUP symbol VGx4 indicates that the instruction operates on four ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This is an alias of [MOVA \(array to vector, four registers\)](#). This means:

- The encodings in this description are named to match the encodings of [MOVA \(array to vector, four registers\)](#).
- The description of [MOVA \(array to vector, four registers\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	Rv		0	1	1	0	0	off3			Zd		0		0

MOV { <Zd1>.D-<Zd4>.D }, ZA.D[<Wv>, <offs>{, VGx4}]

is equivalent to

MOVA { <Zd1>.D-<Zd4>.D }, ZA.D[<Wv>, <offs>{, VGx4}]

and is always the preferred disassembly.

Assembler Symbols

- <Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

Operation

The description of [MOVA \(array to vector, four registers\)](#) gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# MOV (array to vector, two registers)

Move two ZA single-vector groups to two vector registers

The instruction operates on two ZA single-vector groups. The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

The VECTOR GROUP symbol VGx2 indicates that the instruction operates on two ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This is an alias of [MOVA \(array to vector, two registers\)](#). This means:

- The encodings in this description are named to match the encodings of [MOVA \(array to vector, two registers\)](#).
- The description of [MOVA \(array to vector, two registers\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	Rv		0	1	0	0	0	0	off3		Zd			0	

MOV { <Zd1>.D-<Zd2>.D }, ZA.D[<Wv>, <offs>{, VGx2}]

is equivalent to

MOVA { <Zd1>.D-<Zd2>.D }, ZA.D[<Wv>, <offs>{, VGx2}]

and is always the preferred disassembly.

## Assembler Symbols

- <Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

## Operation

The description of [MOVA \(array to vector, two registers\)](#) gives the operational pseudocode for this instruction.

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## MOV (tile to vector, four registers)

Move four ZA tile slices to four vector registers

The instruction operates on four consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 4 in the range 0 to the number of elements in a 128-bit vector segment minus 4.

This instruction is unpredicated.

This is an alias of [MOVA \(tile to vector, four registers\)](#). This means:

- The encodings in this description are named to match the encodings of [MOVA \(tile to vector, four registers\)](#).
- The description of [MOVA \(tile to vector, four registers\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from 4 classes: [8-bit](#) , [16-bit](#) , [32-bit](#) and [64-bit](#)

### 8-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	V	Rs	0	0	1	0	0	0	0	off2		Zd		0	0	

size<1>size<0>

MOV { <Zd1>.B-<Zd4>.B }, ZA0<HV>.B[<Ws>, <offs1>:<offs4>]

is equivalent to

MOVA { <Zd1>.B-<Zd4>.B }, ZA0<HV>.B[<Ws>, <offs1>:<offs4>]

and is always the preferred disassembly.

### 16-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	1	1	0	V	Rs	0	0	1	0	0	0	0	ZAn	o1		Zd		0	0

size<1>size<0>

MOV { <Zd1>.H-<Zd4>.H }, <ZAn><HV>.H[<Ws>, <offs1>:<offs4>]

is equivalent to

MOVA { <Zd1>.H-<Zd4>.H }, <ZAn><HV>.H[<Ws>, <offs1>:<offs4>]

and is always the preferred disassembly.

### 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	1	1	0	V	Rs	0	0	1	0	0	0	0	ZAn		Zd		0	0	

size<1>size<0>

MOV { <Zd1>.S-<Zd4>.S }, <ZAn><HV>.S[<Ws>, <offs1>:<offs4>]

is equivalent to

MOVA { <Zd1>.S-<Zd4>.S }, <ZAn><HV>.S[<Ws>, <offs1>:<offs4>]

and is always the preferred disassembly.

## 64-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	1	1	0	V	Rs	0	0	1	0	0	ZAn	Zd	0	0	0	0	0	0	
								size<1>		size<0>																					

MOV { <Zd1>.D-<Zd4>.D }, <ZAn><HV>.D[<Ws>, <offs1>:<offs4>]

is equivalent to

MOVA { <Zd1>.D-<Zd4>.D }, <ZAn><HV>.D[<Ws>, <offs1>:<offs4>]

and is always the preferred disassembly.

## Assembler Symbols

- <Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <ZAn> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.  
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.  
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.
- <HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs1> For the 8-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "off2" field times 4.  
For the 16-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "o1" field times 4.  
For the 32-bit and 64-bit variant: is the slice index offset, pointing to first of four consecutive slices, with implicit value 0.
- <offs4> For the 8-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "off2" field times 4 plus 3.  
For the 16-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "o1" field times 4 plus 3.  
For the 32-bit and 64-bit variant: is the slice index offset, pointing to last of four consecutive slices, with implicit value 3.

## Operation

The description of [MOVA \(tile to vector, four registers\)](#) gives the operational pseudocode for this instruction.

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

- The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## MOV (tile to vector, single)

Move ZA tile slice to vector register

The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

Inactive elements in the destination vector remain unmodified.

This is an alias of [MOVA \(tile to vector, single\)](#). This means:

- The encodings in this description are named to match the encodings of [MOVA \(tile to vector, single\)](#).
- The description of [MOVA \(tile to vector, single\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from 5 classes: [8-bit](#) , [16-bit](#) , [32-bit](#) , [64-bit](#) and [128-bit](#)

### 8-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	V	Rs		Pg		0				off4					Zd	
size<1>size<0>																Q															

MOV <Zd>.B, <Pg>/M, ZA0<HV>.B[<Ws>, <offs>]

is equivalent to

MOVA <Zd>.B, <Pg>/M, ZA0<HV>.B[<Ws>, <offs>]

and is always the preferred disassembly.

### 16-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	0	1	0	V	Rs		Pg		0	ZAn		off3					Zd		
size<1>size<0>																Q															

MOV <Zd>.H, <Pg>/M, <ZAn><HV>.H[<Ws>, <offs>]

is equivalent to

MOVA <Zd>.H, <Pg>/M, <ZAn><HV>.H[<Ws>, <offs>]

and is always the preferred disassembly.

### 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	V	Rs		Pg		0	ZAn		off2					Zd		
size<1>size<0>																Q															

MOV <Zd>.S, <Pg>/M, <ZAn><HV>.S[<Ws>, <offs>]

is equivalent to

MOVA <Zd>.S, <Pg>/M, <ZAn><HV>.S[<Ws>, <offs>]

and is always the preferred disassembly.

## 64-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	0	1	0	V	Rs		Pg		0		ZAn	o1					Zd		
size<1>size<0>								Q																							

MOV <Zd>.D, <Pg>/M, <ZAn><HV>.D[<Ws>, <offs>]

is equivalent to

MOVA <Zd>.D, <Pg>/M, <ZAn><HV>.D[<Ws>, <offs>]

and is always the preferred disassembly.

## 128-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	0	1	1	V	Rs		Pg		0		ZAn						Zd		
size<1>size<0>								Q																							

MOV <Zd>.Q, <Pg>/M, <ZAn><HV>.Q[<Ws>, <offs>]

is equivalent to

MOVA <Zd>.Q, <Pg>/M, <ZAn><HV>.Q[<Ws>, <offs>]

and is always the preferred disassembly.

## Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <ZAn> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.  
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.  
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.  
For the 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAn" field.

- <HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "off4" field.  
For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "off3" field.  
For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "off2" field.  
For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "o1" field.  
For the 128-bit variant: is the slice index offset 0.



## Operation

The description of [MOVA \(tile to vector, single\)](#) gives the operational pseudocode for this instruction.

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## MOV (tile to vector, two registers)

Move two ZA tile slices to two vector registers

The instruction operates on two consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 2 in the range 0 to the number of elements in a 128-bit vector segment minus 2.

This instruction is unpredicated.

This is an alias of [MOVA \(tile to vector, two registers\)](#). This means:

- The encodings in this description are named to match the encodings of [MOVA \(tile to vector, two registers\)](#).
- The description of [MOVA \(tile to vector, two registers\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from 4 classes: [8-bit](#) , [16-bit](#) , [32-bit](#) and [64-bit](#)

### 8-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	V	Rs	0	0	0	0	0	0	0	off3			Zd		0	
size<1>size<0>																															

MOV { <Zd1>.B-<Zd2>.B }, ZA0<HV>.B[<Ws>, <offs1>:<offs2>]

is equivalent to

MOVA { <Zd1>.B-<Zd2>.B }, ZA0<HV>.B[<Ws>, <offs1>:<offs2>]

and is always the preferred disassembly.

### 16-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	1	1	0	V	Rs	0	0	0	0	0	0	ZAn	off2			Zd			0
size<1>size<0>																															

MOV { <Zd1>.H-<Zd2>.H }, <ZAn><HV>.H[<Ws>, <offs1>:<offs2>]

is equivalent to

MOVA { <Zd1>.H-<Zd2>.H }, <ZAn><HV>.H[<Ws>, <offs1>:<offs2>]

and is always the preferred disassembly.

### 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	1	1	0	V	Rs	0	0	0	0	0	0	ZAn	o1			Zd			0
size<1>size<0>																															

MOV { <Zd1>.S-<Zd2>.S }, <ZAn><HV>.S[<Ws>, <offs1>:<offs2>]

is equivalent to

MOVA { <Zd1>.S-<Zd2>.S }, <ZAn><HV>.S[<Ws>, <offs1>:<offs2>]

and is always the preferred disassembly.

## 64-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	1	1	0	V	Rs	0	0	0	0	0	0	ZAn				Zd		0	
								size<1>		size<0>																					

MOV { <Zd1>.D-<Zd2>.D }, <ZAn><HV>.D[<Ws>, <offs1>:<offs2>]

is equivalent to

MOVA { <Zd1>.D-<Zd2>.D }, <ZAn><HV>.D[<Ws>, <offs1>:<offs2>]

and is always the preferred disassembly.

## Assembler Symbols

- <Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <ZAn> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.  
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.  
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.
- <HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs1> For the 8-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off3" field times 2.  
For the 16-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off2" field times 2.  
For the 32-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "o1" field times 2.  
For the 64-bit variant: is the slice index offset, pointing to first of two consecutive slices, with implicit value 0.
- <offs2> For the 8-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off3" field times 2 plus 1.  
For the 16-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off2" field times 2 plus 1.  
For the 32-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "o1" field times 2 plus 1.  
For the 64-bit variant: is the slice index offset, pointing to last of two consecutive slices, with implicit value 1.

## Operation

The description of [MOVA \(tile to vector, two registers\)](#) gives the operational pseudocode for this instruction.

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (vector to array, four registers)

Move four vector registers to four ZA single-vector groups

The instruction operates on four ZA single-vector groups. The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

The VECTOR GROUP symbol VGx4 indicates that the instruction operates on four ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This is an alias of [MOVA \(vector to array, four registers\)](#). This means:

- The encodings in this description are named to match the encodings of [MOVA \(vector to array, four registers\)](#).
- The description of [MOVA \(vector to array, four registers\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	Rv		0	1	1		Zn		0	0	0	0	0		off3

MOV ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.D-<Zn4>.D }

is equivalent to

MOVA ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.D-<Zn4>.D }

and is always the preferred disassembly.

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

The description of [MOVA \(vector to array, four registers\)](#) gives the operational pseudocode for this instruction.

Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV (vector to array, two registers)

Move two vector registers to two ZA single-vector groups

The instruction operates on two ZA single-vector groups. The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

The VECTOR GROUP symbol VGx2 indicates that the instruction operates on two ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This is an alias of [MOVA \(vector to array, two registers\)](#). This means:

- The encodings in this description are named to match the encodings of [MOVA \(vector to array, two registers\)](#).
- The description of [MOVA \(vector to array, two registers\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	Rv		0	1	0	Zn				0	0	0	off3		

MOV ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.D-<Zn2>.D }

is equivalent to

MOVA ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.D-<Zn2>.D }

and is always the preferred disassembly.

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

The description of [MOVA \(vector to array, two registers\)](#) gives the operational pseudocode for this instruction.

Operational information

- If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

## MOV (vector to tile, four registers)

Move four vector registers to four ZA tile slices

The instruction operates on four consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 4 in the range 0 to the number of elements in a 128-bit vector segment minus 4.

This instruction is unpredicated.

This is an alias of [MOVA \(vector to tile, four registers\)](#). This means:

- The encodings in this description are named to match the encodings of [MOVA \(vector to tile, four registers\)](#).
- The description of [MOVA \(vector to tile, four registers\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from 4 classes: [8-bit](#) , [16-bit](#) , [32-bit](#) and [64-bit](#)

### 8-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	V	Rs	0	0	1	Zn	0	0	0	0	0	0	0	0	off2	
size<1>size<0>																															

MOV ZA0<HV>.B[<Ws>, <offs1>:<offs4>], { <Zn1>.B-<Zn4>.B }

is equivalent to

MOVA ZA0<HV>.B[<Ws>, <offs1>:<offs4>], { <Zn1>.B-<Zn4>.B }

and is always the preferred disassembly.

### 16-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	V	Rs	0	0	1	Zn	0	0	0	0	0	0	0	0	ZAd	o1
size<1>size<0>																															

MOV <ZAd><HV>.H[<Ws>, <offs1>:<offs4>], { <Zn1>.H-<Zn4>.H }

is equivalent to

MOVA <ZAd><HV>.H[<Ws>, <offs1>:<offs4>], { <Zn1>.H-<Zn4>.H }

and is always the preferred disassembly.

### 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	1	0	0	V	Rs	0	0	1	Zn	0	0	0	0	0	0	0	0	0	ZAd
size<1>size<0>																															

MOV <ZAd><HV>.S[<Ws>, <offs1>:<offs4>], { <Zn1>.S-<Zn4>.S }

is equivalent to

MOVA <ZAd><HV>.S[<Ws>, <offs1>:<offs4>], { <Zn1>.S-<Zn4>.S }

and is always the preferred disassembly.

## 64-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	1	0	0	V	Rs	0	0	1	Zn	0	0	0	0	0	0	0	ZAd		

size<1>size<0>

MOV <ZAd><HV>.D[<Ws>, <offs1>:<offs4>], { <Zn1>.D-<Zn4>.D }

is equivalent to

MOVA <ZAd><HV>.D[<Ws>, <offs1>:<offs4>], { <Zn1>.D-<Zn4>.D }

and is always the preferred disassembly.

## Assembler Symbols

<ZAd> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAd" field.  
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAd" field.  
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAd" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs1> For the 8-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "off2" field times 4.

For the 16-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "o1" field times 4.

For the 32-bit and 64-bit variant: is the slice index offset, pointing to first of four consecutive slices, with implicit value 0.

<offs4> For the 8-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "off2" field times 4 plus 3.

For the 16-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "o1" field times 4 plus 3.

For the 32-bit and 64-bit variant: is the slice index offset, pointing to last of four consecutive slices, with implicit value 3.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

## Operation

The description of [MOVA \(vector to tile, four registers\)](#) gives the operational pseudocode for this instruction.

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.



- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## MOV (vector to tile, single)

Move vector register to ZA tile slice

The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

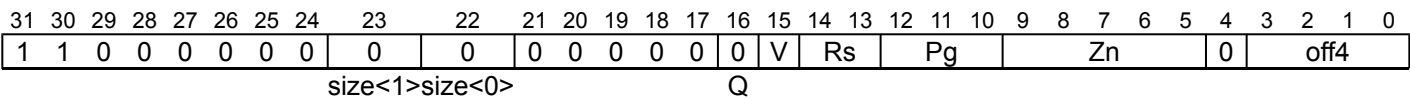
Inactive elements in the destination slice remain unmodified.

This is an alias of [MOVA \(vector to tile, single\)](#). This means:

- The encodings in this description are named to match the encodings of [MOVA \(vector to tile, single\)](#).
- The description of [MOVA \(vector to tile, single\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from 5 classes: [8-bit](#) , [16-bit](#) , [32-bit](#) , [64-bit](#) and [128-bit](#)

### 8-bit



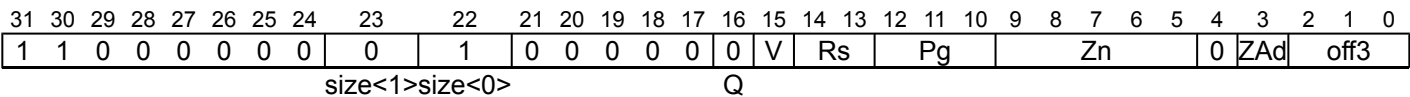
MOV ZA0<HV>.B[<Ws>, <offs>], <Pg>/M, <Zn>.B

is equivalent to

MOVA ZA0<HV>.B[<Ws>, <offs>], <Pg>/M, <Zn>.B

and is always the preferred disassembly.

### 16-bit



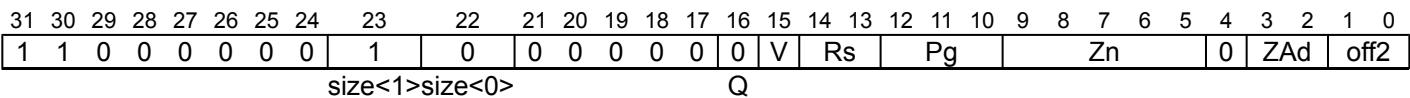
MOV <ZAd><HV>.H[<Ws>, <offs>], <Pg>/M, <Zn>.H

is equivalent to

MOVA <ZAd><HV>.H[<Ws>, <offs>], <Pg>/M, <Zn>.H

and is always the preferred disassembly.

### 32-bit



MOV <ZAd><HV>.S[<Ws>, <offs>], <Pg>/M, <Zn>.S

is equivalent to

MOVA <ZAd><HV>.S[<Ws>, <offs>], <Pg>/M, <Zn>.S

and is always the preferred disassembly.

## 64-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	V	Rs		Pg					Zn		0		ZAd		o1	
size<1>size<0>								Q																							

MOV <ZAd><HV>.D[<Ws>, <offs>], <Pg>/M, <Zn>.D

is equivalent to

MOVA <ZAd><HV>.D[<Ws>, <offs>], <Pg>/M, <Zn>.D

and is always the preferred disassembly.

## 128-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	1	V	Rs		Pg							Zn		0		ZAd	
size<1>size<0>								Q																							

MOV <ZAd><HV>.Q[<Ws>, <offs>], <Pg>/M, <Zn>.Q

is equivalent to

MOVA <ZAd><HV>.Q[<Ws>, <offs>], <Pg>/M, <Zn>.Q

and is always the preferred disassembly.

## Assembler Symbols

<ZAd> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAd" field.  
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAd" field.  
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAd" field.  
For the 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAd" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "off4" field.  
For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "off3" field.  
For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "off2" field.  
For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "o1" field.  
For the 128-bit variant: is the slice index offset 0.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

## Operation

The description of [MOVA \(vector to tile, single\)](#) gives the operational pseudocode for this instruction.

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## MOV (vector to tile, two registers)

Move two vector registers to two ZA tile slices

The instruction operates on two consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 2 in the range 0 to the number of elements in a 128-bit vector segment minus 2.

This instruction is unpredicated.

This is an alias of [MOVA \(vector to tile, two registers\)](#). This means:

- The encodings in this description are named to match the encodings of [MOVA \(vector to tile, two registers\)](#).
- The description of [MOVA \(vector to tile, two registers\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from 4 classes: [8-bit](#) , [16-bit](#) , [32-bit](#) and [64-bit](#)

### 8-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	V	Rs	0	0	0	Zn			0		0	0	off3			
size<1>size<0>																															

MOV ZA0<HV>.B[<Ws>, <offs1>:<offs2>], { <Zn1>.B-<Zn2>.B }

is equivalent to

MOVA ZA0<HV>.B[<Ws>, <offs1>:<offs2>], { <Zn1>.B-<Zn2>.B }

and is always the preferred disassembly.

### 16-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	V	Rs	0	0	0	Zn			0			0	0	ZAd	off2	
size<1>size<0>																															

MOV <ZAd><HV>.H[<Ws>, <offs1>:<offs2>], { <Zn1>.H-<Zn2>.H }

is equivalent to

MOVA <ZAd><HV>.H[<Ws>, <offs1>:<offs2>], { <Zn1>.H-<Zn2>.H }

and is always the preferred disassembly.

### 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	1	0	0	V	Rs	0	0	0	Zn	0	0	0	0	0	0	0	ZAd	o1		
size<1>size<0>																															

MOV <ZAd><HV>.S[<Ws>, <offs1>:<offs2>], { <Zn1>.S-<Zn2>.S }

is equivalent to

MOVA <ZAd><HV>.S[<Ws>, <offs1>:<offs2>], { <Zn1>.S-<Zn2>.S }

and is always the preferred disassembly.

## 64-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	1	0	0	V	Rs	0	0	0			Zn		0	0	0		ZAd		
								size<1>		size<0>																					

MOV <ZAd><HV>.D[<Ws>, <offs1>:<offs2>], { <Zn1>.D-<Zn2>.D }

is equivalent to

MOVA <ZAd><HV>.D[<Ws>, <offs1>:<offs2>], { <Zn1>.D-<Zn2>.D }

and is always the preferred disassembly.

## Assembler Symbols

<ZAd> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAd" field.  
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAd" field.  
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAd" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs1> For the 8-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off3" field times 2.  
For the 16-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off2" field times 2.  
For the 32-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "o1" field times 2.  
For the 64-bit variant: is the slice index offset, pointing to first of two consecutive slices, with implicit value 0.

<offs2> For the 8-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off3" field times 2 plus 1.  
For the 16-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off2" field times 2 plus 1.  
For the 32-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "o1" field times 2 plus 1.  
For the 64-bit variant: is the slice index offset, pointing to last of two consecutive slices, with implicit value 1.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

## Operation

The description of [MOVA \(vector to tile, two registers\)](#) gives the operational pseudocode for this instruction.

## Operational information

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## MOVA (array to vector, four registers)

Move four ZA single-vector groups to four vector registers

The instruction operates on four ZA single-vector groups. The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

The VECTOR GROUP symbol VGx4 indicates that the instruction operates on four ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This instruction is used by the alias [MOV \(array to vector, four registers\)](#).

### SME2

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	Rv	0	1	1	0	0	off3			Zd			0	0	

```
MOVA { <Zd1>.D-<Zd4>.D }, ZA.D[<Wv>, <offs>{, VGx4}]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer offset = UInt(off3);
integer d = UInt(Zd:'00');
constant integer nreg = 4;
```

### Assembler Symbols

- <Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

### Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;

for r = 0 to nreg-1
    bits(VL) result = ZAvector[vec, VL];
    Z[d + r, VL] = result;
    vec = vec + vstride;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.





# MOVA (array to vector, two registers)

Move two ZA single-vector groups to two vector registers

The instruction operates on two ZA single-vector groups. The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

The VECTOR GROUP symbol VGx2 indicates that the instruction operates on two ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This instruction is used by the alias [MOV \(array to vector, two registers\)](#).

## SME2 (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	Rv	0	1	0	0	0	0	off3			Zd			0	

```
MOVA { <Zd1>.D-<Zd2>.D }, ZA.D[<Wv>, <offs>{, VGx2}]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer offset = UInt(off3);
integer d = UInt(Zd:'0');
constant integer nreg = 2;
```

## Assembler Symbols

- <Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;

for r = 0 to nreg-1
    bits(VL) result = ZAvector[vec, VL];
    Z[d + r, VL] = result;
    vec = vec + vstride;
```

## Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



## MOVA (tile to vector, four registers)

Move four ZA tile slices to four vector registers

The instruction operates on four consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 4 in the range 0 to the number of elements in a 128-bit vector segment minus 4.

This instruction is unpredicated.

This instruction is used by the alias [MOV \(tile to vector, four registers\)](#).

It has encodings from 4 classes: [8-bit](#) , [16-bit](#) , [32-bit](#) and [64-bit](#)

### 8-bit

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	V	Rs	0	0	1	0	0	0	0	off2		Zd		0	0	
size<1>size<0>																															

**MOVA** { <Zd1>.B-<Zd4>.B }, ZA0<HV>.B[<Ws>, <offs1>:<offs4>]

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 4;
constant integer esize = 8;
integer d = UInt(Zd:'00');
integer n = 0;
integer offset = UInt(off2:'00');
boolean vertical = V == '1';
```

### 16-bit

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	1	1	0	V	Rs	0	0	1	0	0	0	0	ZAn	o1		Zd		0	0
size<1>size<0>																															

**MOVA** { <Zd1>.H-<Zd4>.H }, <ZAn><HV>.H[<Ws>, <offs1>:<offs4>]

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 4;
constant integer esize = 16;
integer d = UInt(Zd:'00');
integer n = UInt(ZAn);
integer offset = UInt(o1:'00');
boolean vertical = V == '1';
```

### 32-bit

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	1	1	0	V	Rs	0	0	1	0	0	0	0	ZAn		Zd		0	0	
size<1>size<0>																															

MOVA { <Zd1>.S-<Zd4>.S }, <ZAn><HV>.S[<Ws>, <offs1>:<offs4>]

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 4;
constant integer esize = 32;
integer d = UInt(Zd:'00');
integer n = UInt(ZAn);
integer offset = 0;
boolean vertical = V == '1';
```

## 64-bit (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	1	1	0	V	Rs	0	0	1	0	0	ZAn				Zd			0	0

size<1>size<0>

MOVA { <Zd1>.D-<Zd4>.D }, <ZAn><HV>.D[<Ws>, <offs1>:<offs4>]

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 4;
constant integer esize = 64;
integer d = UInt(Zd:'00');
integer n = UInt(ZAn);
integer offset = 0;
boolean vertical = V == '1';
```

## Assembler Symbols

- <Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <ZAn> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.  
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.  
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.
- <HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs1> For the 8-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "off2" field times 4.  
For the 16-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "o1" field times 4.  
For the 32-bit and 64-bit variant: is the slice index offset, pointing to first of four consecutive slices, with implicit value 0.
- <offs4> For the 8-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "off2" field times 4 plus 3.  
For the 16-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "o1" field times 4 plus 3.  
For the 32-bit and 64-bit variant: is the slice index offset, pointing to last of four consecutive slices, with implicit value 3.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
if nreg == 4 && esize == 64 && VL == 128 then UNDEFINED;
integer slices = VL DIV esize;
bits(32) index = X[s, 32];
integer slice = ((UInt(index) - (UInt(index) MOD nreg)) + offset) MOD slices;

for r = 0 to nreg-1
    bits(VL) result = ZAslice[n, esize, vertical, slice + r, VL];
    Z[d + r, VL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## MOVA (tile to vector, single)

Move ZA tile slice to vector register

The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

Inactive elements in the destination vector remain unmodified.

This instruction is used by the alias [MOV \(tile to vector, single\)](#).

It has encodings from 5 classes: [8-bit](#) , [16-bit](#) , [32-bit](#) , [64-bit](#) and [128-bit](#)

### 8-bit

(FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	V	Rs		Pg		0			off4					Zd		
size<1>size<0>																Q															

MOVA <Zd>.B, <Pg>/M, ZA0<HV>.B[<Ws>, <offs>]

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = 0;
integer offset = UInt(off4);
constant integer esize = 8;
integer d = UInt(Zd);
boolean vertical = V == '1';
```

### 16-bit

(FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	0	1	0	V	Rs		Pg		0	ZAn				off3			Zd		
size<1>size<0>																Q															

MOVA <Zd>.H, <Pg>/M, <ZAn><HV>.H[<Ws>, <offs>]

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(ZAn);
integer offset = UInt(off3);
constant integer esize = 16;
integer d = UInt(Zd);
boolean vertical = V == '1';
```

### 32-bit

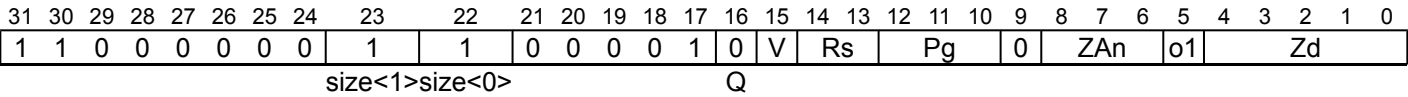
(FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	V	Rs		Pg		0	ZAn							Zd		
size<1>size<0>																Q															

**MOVA** <Zd>.S, <Pg>/M, <ZAn><HV>.S[<Ws>, <offs>]

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(ZAn);
integer offset = UInt(off2);
constant integer esize = 32;
integer d = UInt(Zd);
boolean vertical = V == '1';
```

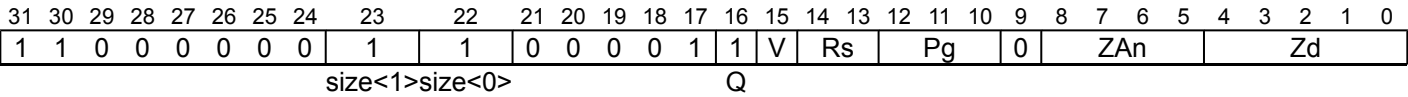
**64-bit**  
**(FEAT\_SME)**



**MOVA** <Zd>.D, <Pg>/M, <ZAn><HV>.D[<Ws>, <offs>]

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(ZAn);
integer offset = UInt(o1);
constant integer esize = 64;
integer d = UInt(Zd);
boolean vertical = V == '1';
```

**128-bit**  
**(FEAT\_SME)**



**MOVA** <Zd>.Q, <Pg>/M, <ZAn><HV>.Q[<Ws>, <offs>]

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(ZAn);
integer offset = 0;
constant integer esize = 128;
integer d = UInt(Zd);
boolean vertical = V == '1';
```

**Assembler Symbols**

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <ZAn> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.  
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.  
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.  
For the 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAn" field.
- <HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V



- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "off4" field.  
 For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "off3" field.  
 For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "off2" field.  
 For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "o1" field.  
 For the 128-bit variant: is the slice index offset 0.

## Operation

```

CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(32) index = X[s, 32];
integer slice = (UInt(index) + offset) MOD dim;
bits(VL) operand = ZAslice[n, esize, vertical, slice, VL];
bits(VL) result = Z[d, VL];

for e = 0 to dim-1
  bits(esize) element = Elem[operand, e, esize];
  if ActivePredicateElement(mask, e, esize) then
    Elem[result, e, esize] = element;

Z[d, VL] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## MOVA (tile to vector, two registers)

Move two ZA tile slices to two vector registers

The instruction operates on two consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 2 in the range 0 to the number of elements in a 128-bit vector segment minus 2.

This instruction is unpredicated.

This instruction is used by the alias [MOV \(tile to vector, two registers\)](#).

It has encodings from 4 classes: [8-bit](#) , [16-bit](#) , [32-bit](#) and [64-bit](#)

### 8-bit

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	V	Rs	0	0	0	0	0	0	off3	Zd			0			
size<1>size<0>																															

**MOVA** { <Zd1>.B-<Zd2>.B }, ZA0<HV>.B[<Ws>, <offs1>:<offs2>]

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 2;
constant integer esize = 8;
integer d = UInt(Zd:'0');
integer n = 0;
integer offset = UInt(off3:'0');
boolean vertical = V == '1';
```

### 16-bit

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	1	1	0	V	Rs	0	0	0	0	0	0	ZAn	off2			Zd			0
size<1>size<0>																															

**MOVA** { <Zd1>.H-<Zd2>.H }, <ZAn><HV>.H[<Ws>, <offs1>:<offs2>]

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 2;
constant integer esize = 16;
integer d = UInt(Zd:'0');
integer n = UInt(ZAn);
integer offset = UInt(off2:'0');
boolean vertical = V == '1';
```

### 32-bit

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	1	1	0	V	Rs	0	0	0	0	0	0	ZAn	o1			Zd			0
size<1>size<0>																															

MOVA { <Zd1>.S-<Zd2>.S }, <ZAn><HV>.S[<Ws>, <offs1>:<offs2>]

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 2;
constant integer esize = 32;
integer d = UInt(Zd:'0');
integer n = UInt(ZAn);
integer offset = UInt(o1:'0');
boolean vertical = V == '1';
```

## 64-bit (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	1	1	0	V	Rs	0	0	0	0	0	0	ZAn				Zd		0	

size<1>size<0>

MOVA { <Zd1>.D-<Zd2>.D }, <ZAn><HV>.D[<Ws>, <offs1>:<offs2>]

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 2;
constant integer esize = 64;
integer d = UInt(Zd:'0');
integer n = UInt(ZAn);
integer offset = 0;
boolean vertical = V == '1';
```

## Assembler Symbols

- <Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <ZAn> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.  
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.  
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.
- <HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs1> For the 8-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off3" field times 2.  
For the 16-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off2" field times 2.  
For the 32-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "o1" field times 2.  
For the 64-bit variant: is the slice index offset, pointing to first of two consecutive slices, with implicit value 0.
- <offs2> For the 8-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off3" field times 2 plus 1.  
For the 16-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off2" field times 2 plus 1.  
For the 32-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "o1" field times 2 plus 1.

For the 64-bit variant: is the slice index offset, pointing to last of two consecutive slices, with implicit value 1.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
if nreg == 4 && esize == 64 && VL == 128 then UNDEFINED;
integer slices = VL DIV esize;
bits(32) index = X[s, 32];
integer slice = ((UInt(index) - (UInt(index) MOD nreg)) + offset) MOD slices;

for r = 0 to nreg-1
    bits(VL) result = ZAslice[n, esize, vertical, slice + r, VL];
    Z[d + r, VL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# MOVA (vector to array, four registers)

Move four vector registers to four ZA single-vector groups

The instruction operates on four ZA single-vector groups. The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

The VECTOR GROUP symbol VGx4 indicates that the instruction operates on four ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This instruction is used by the alias [MOV \(vector to array, four registers\)](#).

## SME2 (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	Rv		0	1	1		Zn		0	0	0	0	0		off3

MOVA ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.D--<Zn4>.D }

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer offset = UInt(off3);
integer n = UInt(Zn:'00');
constant integer nreg = 4;
```

## Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;

for r = 0 to nreg-1
    bits(VL) result = Z[n + r, VL];
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

## Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



## MOVA (vector to array, two registers)

Move two vector registers to two ZA single-vector groups

The instruction operates on two ZA single-vector groups. The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

The VECTOR GROUP symbol VGx2 indicates that the instruction operates on two ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This instruction is used by the alias [MOV \(vector to array, two registers\)](#).

### SME2 (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	Rv		0	1	0		Zn				0	0	0		off3

**MOVA** ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.D-<Zn2>.D }

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer offset = UInt(off3);
integer n = UInt(Zn:'0');
constant integer nreg = 2;
```

### Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

### Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;

for r = 0 to nreg-1
    bits(VL) result = Z[n + r, VL];
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.





## MOVA (vector to tile, four registers)

Move four vector registers to four ZA tile slices

The instruction operates on four consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 4 in the range 0 to the number of elements in a 128-bit vector segment minus 4.

This instruction is unpredicated.

This instruction is used by the alias [MOV \(vector to tile, four registers\)](#).

It has encodings from 4 classes: [8-bit](#) , [16-bit](#) , [32-bit](#) and [64-bit](#)

### 8-bit

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	V	Rs	0	0	1	Zn	0	0	0	0	0	0	0	0	off2	
size<1>size<0>																															

**MOVA** ZA0<HV>.B[<Ws>, <offs1>:<offs4>], { <Zn1>.B-<Zn4>.B }

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 4;
constant integer esize = 8;
integer n = UInt(Zn:'00');
integer d = 0;
integer offset = UInt(off2:'00');
boolean vertical = V == '1';
```

### 16-bit

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	V	Rs	0	0	1	Zn	0	0	0	0	0	0	0	0	ZAd	o1
size<1>size<0>																															

**MOVA** <ZAd><HV>.H[<Ws>, <offs1>:<offs4>], { <Zn1>.H-<Zn4>.H }

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 4;
constant integer esize = 16;
integer n = UInt(Zn:'00');
integer d = UInt(ZAd);
integer offset = UInt(o1:'00');
boolean vertical = V == '1';
```

### 32-bit

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	1	0	0	V	Rs	0	0	1	Zn	0	0	0	0	0	0	0	0	0	ZAd	
size<1>size<0>																															

MOVA <ZAd><HV>.S[<Ws>, <offs1>:<offs4>], { <Zn1>.S-<Zn4>.S }

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 4;
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer d = UInt(ZAd);
integer offset = 0;
boolean vertical = V == '1';
```

## 64-bit (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	1	0	0	V	Rs	0	0	1	Zn	0	0	0	0	0	0	0	ZAd		

size<1>size<0>

MOVA <ZAd><HV>.D[<Ws>, <offs1>:<offs4>], { <Zn1>.D-<Zn4>.D }

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 4;
constant integer esize = 64;
integer n = UInt(Zn:'00');
integer d = UInt(ZAd);
integer offset = 0;
boolean vertical = V == '1';
```

## Assembler Symbols

<ZAd> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAd" field.  
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAd" field.  
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAd" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs1> For the 8-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "off2" field times 4.

For the 16-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "o1" field times 4.

For the 32-bit and 64-bit variant: is the slice index offset, pointing to first of four consecutive slices, with implicit value 0.

<offs4> For the 8-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "off2" field times 4 plus 3.

For the 16-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "o1" field times 4 plus 3.

For the 32-bit and 64-bit variant: is the slice index offset, pointing to last of four consecutive slices, with implicit value 3.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

## Operation

```
CheckStreamingSVEAndZAAEnabled\(\);  
constant integer VL = CurrentVL;  
if nreg == 4 && esize == 64 && VL == 128 then UNDEFINED;  
integer slices = VL DIV esize;  
bits(32) index = X[s, 32];  
integer slice = ((UInt(index) - (UInt(index) MOD nreg)) + offset) MOD slices;  
  
for r = 0 to nreg-1  
    bits(VL) result = Z[n + r, VL];  
    ZAslice[d, esize, vertical, slice + r, VL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## MOVA (vector to tile, single)

Move vector register to ZA tile slice

The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

Inactive elements in the destination slice remain unmodified.

This instruction is used by the alias [MOV \(vector to tile, single\)](#).

It has encodings from 5 classes: [8-bit](#) , [16-bit](#) , [32-bit](#) , [64-bit](#) and [128-bit](#)

### 8-bit

(FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	V	Rs		Pg								Zn		0		off4
size<1>size<0>																Q															

**MOVA** **ZA0<HV>.B**[<Ws>, <offs>], <Pg>/M, <Zn>.B

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(Zn);
integer d = 0;
integer offset = UInt(off4);
constant integer esize = 8;
boolean vertical = V == '1';
```

### 16-bit

(FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	V	Rs		Pg						Zn			0	ZAd		off3
size<1>size<0>																Q															

**MOVA** <ZAd><HV>.H[<Ws>, <offs>], <Pg>/M, <Zn>.H

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(Zn);
integer d = UInt(ZAd);
integer offset = UInt(off3);
constant integer esize = 16;
boolean vertical = V == '1';
```

### 32-bit

(FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	V	Rs		Pg									0	ZAd	off2	
size<1>size<0>																Q															

**MOVA** <ZAd><HV>.S[<Ws>, <offs>], <Pg>/M, <Zn>.S

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(Zn);
integer d = UInt(ZAd);
integer offset = UInt(off2);
constant integer esize = 32;
boolean vertical = V == '1';
```

## 64-bit (FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	V	Rs		Pg								0		ZAd		o1
size<1>size<0>																Q															

**MOVA** <ZAd><HV>.D[<Ws>, <offs>], <Pg>/M, <Zn>.D

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(Zn);
integer d = UInt(ZAd);
integer offset = UInt(o1);
constant integer esize = 64;
boolean vertical = V == '1';
```

## 128-bit (FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	1	V	Rs	Pg			Zn				0	ZAd					
size<1>size<0>																Q															

**MOVA** <ZAd><HV>.Q[<Ws>, <offs>], <Pg>/M, <Zn>.Q

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(Zn);
integer d = UInt(ZAd);
integer offset = 0;
constant integer esize = 128;
boolean vertical = V == '1';
```

## Assembler Symbols

- <ZAd> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAd" field.  
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAd" field.  
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAd" field.  
For the 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAd" field.

- <HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

- <offs> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "off4" field.

For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "off3" field.  
 For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "off2" field.  
 For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "o1" field.  
 For the 128-bit variant: is the slice index offset 0.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.  
 <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = Z[n, VL];
bits(32) index = X[s, 32];
integer slice = (UInt(index) + offset) MOD dim;
bits(VL) result = ZAslice[d, esize, vertical, slice, VL];

for e = 0 to dim-1
  bits(esize) element = Elem[operand, e, esize];
  if ActivePredicateElement(mask, e, esize) then
    Elem[result, e, esize] = element;

ZAslice[d, esize, vertical, slice, VL] = result;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## MOVA (vector to tile, two registers)

Move two vector registers to two ZA tile slices

The instruction operates on two consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 2 in the range 0 to the number of elements in a 128-bit vector segment minus 2.

This instruction is unpredicated.

This instruction is used by the alias [MOV \(vector to tile, two registers\)](#).

It has encodings from 4 classes: [8-bit](#) , [16-bit](#) , [32-bit](#) and [64-bit](#)

### 8-bit

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	V	Rs	0	0	0	Zn			0	0	0	off3					
size<1>size<0>																															

**MOVA** **ZA0**<HV>.B[<Ws>, <offs1>:<offs2>], { <Zn1>.B-<Zn2>.B }

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 2;
constant integer esize = 8;
integer n = UInt(Zn:'0');
integer d = 0;
integer offset = UInt(off3:'0');
boolean vertical = V == '1';
```

### 16-bit

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	V	Rs	0	0	0	Zn			0	0	0	ZAd	off2			
size<1>size<0>																															

**MOVA** <ZAd><HV>.H[<Ws>, <offs1>:<offs2>], { <Zn1>.H-<Zn2>.H }

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 2;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(ZAd);
integer offset = UInt(off2:'0');
boolean vertical = V == '1';
```

### 32-bit

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	1	0	0	V	Rs	0	0	0	Zn			0	0	0	ZAd	o1				
size<1>size<0>																															

**MOVA** <ZAd><HV>.S[<Ws>, <offs1>:<offs2>], { <Zn1>.S-<Zn2>.S }

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 2;
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer d = UInt(ZAd);
integer offset = UInt(o1:'0');
boolean vertical = V == '1';
```

## 64-bit (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	1	0	0	V	Rs	0	0	0			Zn		0	0	0		ZAd		

size<1>size<0>

**MOVA** <ZAd><HV>.D[<Ws>, <offs1>:<offs2>], { <Zn1>.D-<Zn2>.D }

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 2;
constant integer esize = 64;
integer n = UInt(Zn:'0');
integer d = UInt(ZAd);
integer offset = 0;
boolean vertical = V == '1';
```

## Assembler Symbols

<ZAd> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAd" field.  
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAd" field.  
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAd" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs1> For the 8-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off3" field times 2.

For the 16-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off2" field times 2.

For the 32-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "o1" field times 2.

For the 64-bit variant: is the slice index offset, pointing to first of two consecutive slices, with implicit value 0.

<offs2> For the 8-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off3" field times 2 plus 1.

For the 16-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off2" field times 2 plus 1.

For the 32-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "o1" field times 2 plus 1.

For the 64-bit variant: is the slice index offset, pointing to last of two consecutive slices, with implicit value 1.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.



<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
if nreg == 4 && esize == 64 && VL == 128 then UNDEFINED;
integer slices = VL DIV esize;
bits(32) index = X[s, 32];
integer slice = ((UInt(index) - (UInt(index) MOD nreg)) + offset) MOD slices;

for r = 0 to nreg-1
    bits(VL) result = Z[n + r, VL];
    ZAslice[d, esize, vertical, slice + r, VL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# MOVAZ (array to vector, four registers)

Move and zero four ZA single-vector groups to vector registers

The instruction operates on four ZA single-vector groups. The ZA single-vector groups are zeroed after moving their contents to the destination vectors. The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

The VECTOR GROUP symbol VGx4 indicates that the instruction operates on four ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

## SME2 (FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	Rv		0	1	1	1	0		off3			Zd			0	0

MOVAZ { <Zd1>.D-<Zd4>.D }, ZA.D[<Wv>, <offs>{, VGx4}]

```
if !HaveSME2p1() then UNDEFINED;
integer v = UInt('010':Rv);
integer offset = UInt(off3);
integer d = UInt(Zd:'00');
constant integer nreg = 4;
```

## Assembler Symbols

- <Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;

for r = 0 to nreg-1
    bits(VL) result = ZAvector[vec, VL];
    ZAvector[vec, VL] = Zeros(VL);
    Z[d + r, VL] = result;
    vec = vec + vstride;
```

## Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# MOVAZ (array to vector, two registers)

Move and zero two ZA single-vector groups to vector registers

The instruction operates on two ZA single-vector groups. The ZA single-vector groups are zeroed after moving their contents to the destination vectors. The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

The VECTOR GROUP symbol VGx2 indicates that the instruction operates on two ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

## SME2 (FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	Rv	0	1	0	1	0	off3	Zd			0				

MOVAZ { <Zd1>.D-<Zd2>.D }, ZA.D[<Wv>, <offs>{, VGx2}]

```
if !HaveSME2p1() then UNDEFINED;
integer v = UInt('010':Rv);
integer offset = UInt(off3);
integer d = UInt(Zd:'0');
constant integer nreg = 2;
```

## Assembler Symbols

- <Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;

for r = 0 to nreg-1
    bits(VL) result = ZAvector[vec, VL];
    ZAvector[vec, VL] = Zeros(VL);
    Z[d + r, VL] = result;
    vec = vec + vstride;
```

## Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## MOVAZ (tile to vector, four registers)

Move and zero four ZA tile slices to vector registers

The instruction operates on four consecutive horizontal or vertical slices within a named ZA tile of the specified element size. The tile slices are zeroed after moving their contents to the destination vectors.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 4 in the range 0 to the number of elements in a 128-bit vector segment minus 4.

This instruction is unpredicated.

It has encodings from 4 classes: [8-bit](#) , [16-bit](#) , [32-bit](#) and [64-bit](#)

### 8-bit (FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	V	Rs	0	0	1	1	0	0	0	off2	Zd	0	0	0	0	
size<1>size<0>																															

MOVAZ { <Zd1>.B-<Zd4>.B }, ZA0<HV>.B[<Ws>, <offs1>:<offs4>]

```
if !HaveSME2p1() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 4;
constant integer esize = 8;
integer d = UInt(Zd:'00');
integer n = 0;
integer offset = UInt(off2:'00');
boolean vertical = V == '1';
```

### 16-bit (FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	1	1	0	V	Rs	0	0	1	1	0	0	0	ZAn	o1	Zd	0	0	0	0
size<1>size<0>																															

MOVAZ { <Zd1>.H-<Zd4>.H }, <ZAn><HV>.H[<Ws>, <offs1>:<offs4>]

```
if !HaveSME2p1() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 4;
constant integer esize = 16;
integer d = UInt(Zd:'00');
integer n = UInt(ZAn);
integer offset = UInt(o1:'00');
boolean vertical = V == '1';
```

### 32-bit (FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	1	1	0	V	Rs	0	0	1	1	0	0	0	ZAn	Zd	0	0	0	0	0
size<1>size<0>																															

MOVAZ { <Zd1>.S-<Zd4>.S }, <ZAn><HV>.S[<Ws>, <offs1>:<offs4>]

```
if !HaveSME2p1() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 4;
constant integer esize = 32;
integer d = UInt(Zd:'00');
integer n = UInt(ZAn);
integer offset = 0;
boolean vertical = V == '1';
```

## 64-bit (FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	1	1	0	V	Rs	0	0	1	1	0	ZAn				Zd			0	0

size<1>size<0>

MOVAZ { <Zd1>.D-<Zd4>.D }, <ZAn><HV>.D[<Ws>, <offs1>:<offs4>]

```
if !HaveSME2p1() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 4;
constant integer esize = 64;
integer d = UInt(Zd:'00');
integer n = UInt(ZAn);
integer offset = 0;
boolean vertical = V == '1';
```

## Assembler Symbols

- <Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <ZAn> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.  
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.  
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.
- <HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs1> For the 8-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "off2" field times 4.  
For the 16-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "o1" field times 4.  
For the 32-bit and 64-bit variant: is the slice index offset, pointing to first of four consecutive slices, with implicit value 0.
- <offs4> For the 8-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "off2" field times 4 plus 3.  
For the 16-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "o1" field times 4 plus 3.  
For the 32-bit and 64-bit variant: is the slice index offset, pointing to last of four consecutive slices, with implicit value 3.

## Operation

```
CheckStreamingSVEAndZAEEnabled();  
constant integer VL = CurrentVL;  
if nreg == 4 && esize == 64 && VL == 128 then UNDEFINED;  
integer slices = VL DIV esize;  
bits(32) index = X[s, 32];  
integer slice = ((UInt(index) - (UInt(index) MOD nreg)) + offset) MOD slices;  
  
for r = 0 to nreg-1  
    bits(VL) result = ZAslice[n, esize, vertical, slice + r, VL];  
    ZAslice[n, esize, vertical, slice + r, VL] = Zeros(VL);  
    Z[d + r, VL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## MOVAZ (tile to vector, single)

Move and zero ZA tile slice to vector register

The instruction operates on a horizontal or vertical slice within a named ZA tile of the specified element size. The tile slice is zeroed after moving its contents to the destination vector.

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

This instruction is unpredicated.

It has encodings from 5 classes: [8-bit](#) , [16-bit](#) , [32-bit](#) , [64-bit](#) and [128-bit](#)

### 8-bit

(FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	V	Rs	0	0	0	1	off4				Zd					
size<1>size<0>																Q															

MOVAZ <Zd>.B, ZA0<HV>.B[<Ws>, <offs>]

```
if !HaveSME2p1() then UNDEFINED;
integer s = UInt('011':Rs);
integer n = 0;
integer offset = UInt(off4);
constant integer esize = 8;
integer d = UInt(Zd);
boolean vertical = V == '1';
```

### 16-bit

(FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	0	1	0	V	Rs	0	0	0	1	ZAn	off3				Zd				
size<1>size<0>																Q															

MOVAZ <Zd>.H, <ZAn><HV>.H[<Ws>, <offs>]

```
if !HaveSME2p1() then UNDEFINED;
integer s = UInt('011':Rs);
integer n = UInt(ZAn);
integer offset = UInt(off3);
constant integer esize = 16;
integer d = UInt(Zd);
boolean vertical = V == '1';
```

### 32-bit

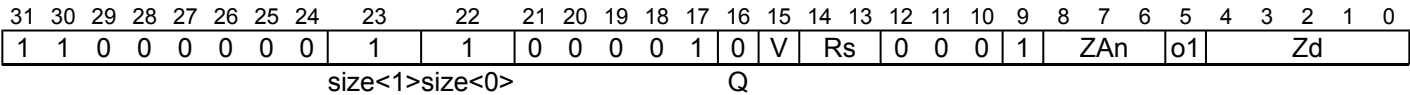
(FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	V	Rs	0	0	0	1	ZAn	off2				Zd				
size<1>size<0>																Q															

MOVAZ <Zd>.S, <ZAn><HV>.S[<Ws>, <offs>]

```
if !HaveSME2p1() then UNDEFINED;
integer s = UInt('011':Rs);
integer n = UInt(ZAn);
integer offset = UInt(off2);
constant integer esize = 32;
integer d = UInt(Zd);
boolean vertical = V == '1';
```

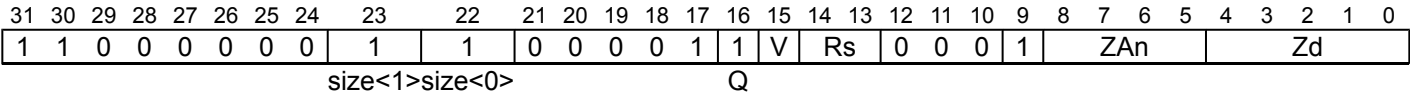
64-bit  
(FEAT\_SME2p1)



MOVAZ <Zd>.D, <ZAn><HV>.D[<Ws>, <offs>]

```
if !HaveSME2p1() then UNDEFINED;
integer s = UInt('011':Rs);
integer n = UInt(ZAn);
integer offset = UInt(o1);
constant integer esize = 64;
integer d = UInt(Zd);
boolean vertical = V == '1';
```

128-bit  
(FEAT\_SME2p1)



MOVAZ <Zd>.Q, <ZAn><HV>.Q[<Ws>, <offs>]

```
if !HaveSME2p1() then UNDEFINED;
integer s = UInt('011':Rs);
integer n = UInt(ZAn);
integer offset = 0;
constant integer esize = 128;
integer d = UInt(Zd);
boolean vertical = V == '1';
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <ZAn> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.  
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.  
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.  
For the 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAn" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "off4" field.  
For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "off3" field.

For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "off2" field.

For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "o1" field.

For the 128-bit variant: is the slice index offset 0.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(32) index = X[s, 32];
integer slice = (UInt(index) + offset) MOD dim;
bits(VL) operand = ZAslice[n, esize, vertical, slice, VL];
ZAslice[n, esize, vertical, slice, VL] = Zeros(VL);
Z[d, VL] = operand;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## MOVAZ (tile to vector, two registers)

Move and zero two ZA tile slices to vector registers

The instruction operates on two consecutive horizontal or vertical slices within a named ZA tile of the specified element size. The tile slices are zeroed after moving their contents to the destination vectors.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 2 in the range 0 to the number of elements in a 128-bit vector segment minus 2.

This instruction is unpredicated.

It has encodings from 4 classes: [8-bit](#) , [16-bit](#) , [32-bit](#) and [64-bit](#)

### 8-bit

(FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	V	Rs	0	0	0	1	0	off3			Zd			0		
size<1>size<0>																															

MOVAZ { <Zd1>.B-<Zd2>.B }, ZA0<HV>.B[<Ws>, <offs1>:<offs2>]

```
if !HaveSME2p1() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 2;
constant integer esize = 8;
integer d = UInt(Zd:'0');
integer n = 0;
integer offset = UInt(off3:'0');
boolean vertical = V == '1';
```

### 16-bit

(FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	0	1	1	0	V	Rs	0	0	0	1	0	ZAn	off2				Zd		0	
size<1>size<0>																															

MOVAZ { <Zd1>.H-<Zd2>.H }, <ZAn><HV>.H[<Ws>, <offs1>:<offs2>]

```
if !HaveSME2p1() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 2;
constant integer esize = 16;
integer d = UInt(Zd:'0');
integer n = UInt(ZAn);
integer offset = UInt(off2:'0');
boolean vertical = V == '1';
```

### 32-bit

(FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	1	1	0	V	Rs	0	0	0	1	0	ZAn	o1			Zd			0	
size<1>size<0>																															

MOVAZ { <Zd1>.S-<Zd2>.S }, <ZAn><HV>.S[<Ws>, <offs1>:<offs2>]

```
if !HaveSME2p1() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 2;
constant integer esize = 32;
integer d = UInt(Zd:'0');
integer n = UInt(ZAn);
integer offset = UInt(o1:'0');
boolean vertical = V == '1';
```

## 64-bit (FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	1	1	0	V	Rs	0	0	0	1	0	ZAn				Zd			0	

size<1>size<0>

MOVAZ { <Zd1>.D-<Zd2>.D }, <ZAn><HV>.D[<Ws>, <offs1>:<offs2>]

```
if !HaveSME2p1() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 2;
constant integer esize = 64;
integer d = UInt(Zd:'0');
integer n = UInt(ZAn);
integer offset = 0;
boolean vertical = V == '1';
```

## Assembler Symbols

- <Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <ZAn> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.  
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.  
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.
- <HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs1> For the 8-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off3" field times 2.  
For the 16-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off2" field times 2.  
For the 32-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "o1" field times 2.  
For the 64-bit variant: is the slice index offset, pointing to first of two consecutive slices, with implicit value 0.
- <offs2> For the 8-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off3" field times 2 plus 1.  
For the 16-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off2" field times 2 plus 1.  
For the 32-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "o1" field times 2 plus 1.

For the 64-bit variant: is the slice index offset, pointing to last of two consecutive slices, with implicit value 1.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
if nreg == 4 && esize == 64 && VL == 128 then UNDEFINED;
integer slices = VL DIV esize;
bits(32) index = X[s, 32];
integer slice = ((UInt(index) - (UInt(index) MOD nreg)) + offset) MOD slices;

for r = 0 to nreg-1
    bits(VL) result = ZAslice[n, esize, vertical, slice + r, VL];
    ZAslice[n, esize, vertical, slice + r, VL] = Zeros(VL);
    Z[d + r, VL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOVN (scalar to ZT0)

Move 8 bytes from general-purpose register to ZT0

Move 8 bytes to the ZT0 register at the byte offset specified by the immediate index from a general-purpose register. This instruction is UNDEFINED in Non-debug state.

SME2  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	1	1	1	0	0	off3			0	0	1	1	1	1	1	Rt				

```
MOVN ZT0[<offs>], <Xt>
```

```
if !HaveSME2() || !Halted() then UNDEFINED;
integer t = UInt(Rt);
integer offset = UInt(off3);
```

Assembler Symbols

- <offs> Is the immediate byte offset, a multiple of 8 in the range of 0 to 56, encoded in the "off3" field as <offs>/8.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Operation

```
CheckSMEEnabled();
CheckSMEZT0Enabled();
bits(512) result = ZT0[512];

Elem[result, offset, 64] = X[t, 64];
ZT0[512] = result;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

MOVN (ZT0 to scalar)

Move 8 bytes from ZT0 to general-purpose register

Move 8 bytes to a general-purpose register from the ZT0 register at the byte offset specified by the immediate index. This instruction is UNDEFINED in Non-debug state.

SME2  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	1	1	0	0	0	off3			0	0	1	1	1	1	1	Rt				

MOVN <Xt>, ZT0[<offs>]

```
if !HaveSME2() || !Halted() then UNDEFINED;
integer t = UInt(Rt);
integer offset = UInt(off3);
```

Assembler Symbols

- <Xt>
- Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <offs>
- Is the immediate byte offset, a multiple of 8 in the range of 0 to 56, encoded in the "off3" field as <offs>/8.

Operation

```
CheckSMEEnabled();
CheckSMEZT0Enabled();
bits(512) operand = ZT0[512];

X[t, 64] = Elem[operand, offset, 64];
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



RDSVL

Read multiple of Streaming SVE vector register size to scalar register

Multiply the Streaming SVE vector register size in bytes by an immediate in the range -32 to 31 and place the result in the 64-bit destination general-purpose register.  
This instruction does not require the PE to be in Streaming SVE mode.

SME  
(FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	0	1	0	1	1	imm6						Rd				

RDSVL <Xd>, #<imm>

```
if !HaveSME() then UNDEFINED;
integer d = UInt(Rd);
integer imm = SInt(imm6);
```

Assembler Symbols

- <Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.
- <imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

```
CheckSMEEnabled();
constant integer SVL = CurrentSVL;
integer len = imm * (SVL DIV 8);
X[d, 64] = len<63:0>;
```

Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

SCLAMP

Multi-vector signed clamp to minimum/maximum vector

Clamp each signed element in the two or four destination vectors to between the signed minimum value in the corresponding element of the first source vector and the signed maximum value in the corresponding element of the second source vector and destructively place the clamped results in the corresponding elements of the two or four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	Zm						1	1	0	0	0	1	Zn						Zd			0
U																															

SCLAMP { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<T>, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd:'0');
constant integer nreg = 2;
```

Four registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	size	1	Zm						1	1	0	0	1	1	Zn						Zd			0	0
U																																

SCLAMP { <Zd1>.<T>-<Zd4>.<T> }, <Zn>.<T>, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd:'00');
constant integer nreg = 4;
```

Assembler Symbols

- <Zd1>

For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.  
  
For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D
- <Zd4>

Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = Z[d+r, VL];
    for e = 0 to elements-1
        integer element1 = SInt(Elem[operand1, e, esize]);
        integer element2 = SInt(Elem[operand2, e, esize]);
        integer element3 = SInt(Elem[operand3, e, esize]);
        integer res = Min(Max(element1, element3), element2);
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SCVTF

Multi-vector signed integer convert to floating-point

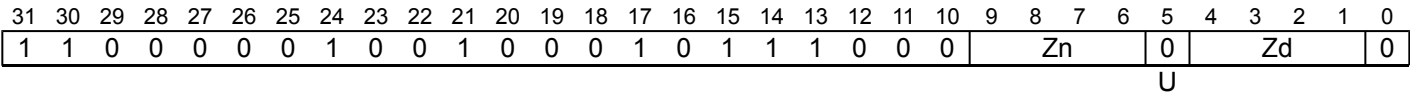
Convert to single-precision from signed 32-bit integer, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

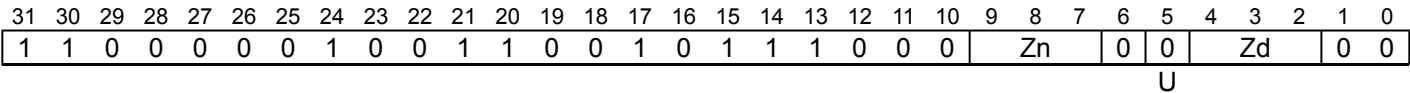
Two registers  
(FEAT\_SME2)



```
SCVTF { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd:'0');
constant integer nreg = 2;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Four registers  
(FEAT\_SME2)



```
SCVTF { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'00');
integer d = UInt(Zd:'00');
constant integer nreg = 4;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

- <Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.  
For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(32) element = Elem[operand, e, 32];
        Elem[results[r], e, 32] = FixedToFP(element, 0, unsigned, FPCR[], rounding, 32);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SDOT (2-way, multiple and indexed vector)

Multi-vector signed integer dot-product by indexed element

The signed integer dot product instruction computes the dot product of two signed 16-bit integer values held in each 32-bit element of the two or four first source vectors and two signed 16-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

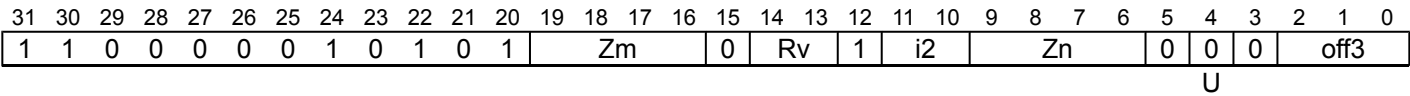
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

### Two ZA single-vectors

(FEAT\_SME2)

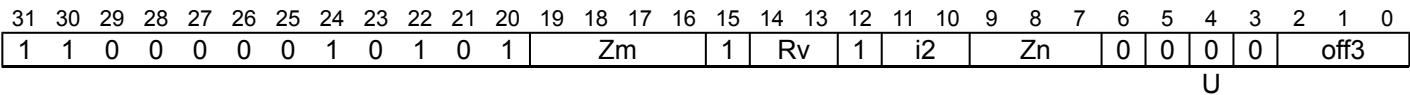


SDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 2;
```

### Four ZA single-vectors

(FEAT\_SME2)



SDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 4;
```

### Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> Is the immediate index of a group of two 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

## Operation

```
CheckStreamingSVEAndZAEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        for i = 0 to 1
            integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = SInt(Elem[operand2, 2 * s + i, esize DIV 2]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SDOT (2-way, multiple and single vector)

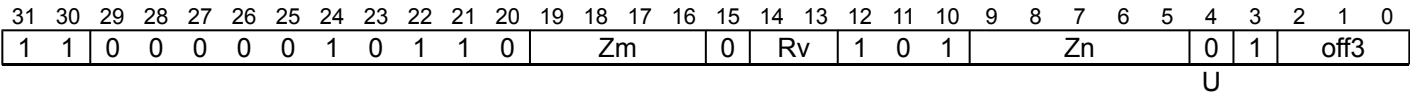
Multi-vector signed integer dot-product by vector

The signed integer dot product instruction computes the dot product of two signed 16-bit integer values held in each 32-bit element of the two or four first source vectors and two signed 16-bit integer values in the corresponding 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

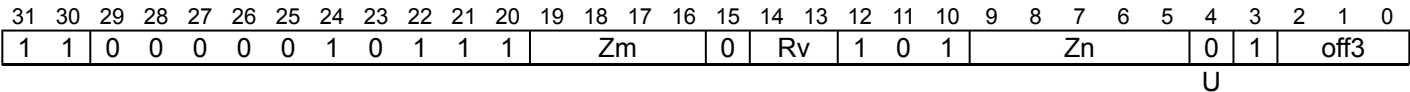
Two ZA single-vectors  
(FEAT\_SME2)



SDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors  
(FEAT\_SME2)



SDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.



## Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 1
            integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = SInt(Elem[operand2, 2 * e + i, esize DIV 2]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SDOT (2-way, multiple vectors)

Multi-vector signed integer dot-product

The signed integer dot product instruction computes the dot product of two signed 16-bit integer values held in each 32-bit element of the two or four first source vectors and two signed 16-bit integer values in the corresponding 32-bit element of the two or four second source vectors. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

### Two ZA single-vectors (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	Zm				0	0	Rv		1	0	1	Zn				0	0	1	off3		
U																															

SDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

### Four ZA single-vectors (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	1	1	Zm				0	1	0	Rv		1	0	1	Zn				0	0	0	1	off3	
U																																

SDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

### Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.
- For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 1
            integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = SInt(Elem[operand2, 2 * e + i, esize DIV 2]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SDOT (4-way, multiple and indexed vector)

Multi-vector signed integer dot-product by indexed element

The signed integer dot product instruction computes the dot product of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four signed 8-bit or 16-bit integer values in the corresponding indexed 32-bit or 64-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 4 classes: [Two ZA single-vectors of 32-bit elements](#) , [Two ZA single-vectors of 64-bit elements](#) , [Four ZA single-vectors of 32-bit elements](#) and [Four ZA single-vectors of 64-bit elements](#)

### Two ZA single-vectors of 32-bit elements (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				0	Rv		1	i2	Zn				1	0	0	off3			
U																															

SDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 2;
```

### Two ZA single-vectors of 64-bit elements (FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				0	Rv		0	0	i1	Zn				0	0	1	off3		
U																															

SDOT ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 64;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i1);
constant integer nreg = 2;
```

## Four ZA single-vectors of 32-bit elements (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	0	1				Zm	1		Rv	1		i2			Zn	0	1	0	0		off3	

U

SDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 4;
```

## Four ZA single-vectors of 64-bit elements (FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1				Zm	1		Rv	0	0	i1			Zn	0	0	0	1		off3	

U

SDOT ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 64;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i1);
constant integer nreg = 4;
```

## Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors of 32-bit elements and two ZA single-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA single-vectors of 32-bit elements and four ZA single-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> For the four ZA single-vectors of 32-bit elements and two ZA single-vectors of 32-bit elements variant: is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.  
For the four ZA single-vectors of 64-bit elements and two ZA single-vectors of 64-bit elements variant: is the immediate index of a 64-bit group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the "i1" field.

## Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        for i = 0 to 3
            integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SDOT (4-way, multiple and single vector)

Multi-vector signed integer dot-product by vector

The signed integer dot product instruction computes the dot product of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four signed 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

Two ZA single-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm			0	Rv		1	0	1	Zn				0	0	off3				
U																															

SDOT ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm			0	Rv		1	0	1	Zn				0	0	off3				
U																															

SDOT ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Tb> Is the size specifier, encoded in "sz":
 

sz	<Tb>
0	B
1	H
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;

```



SDOT (4-way, multiple vectors)

Multi-vector signed integer dot-product

The signed integer dot product instruction computes the dot product of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four signed 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the two or four second source vectors. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

Two ZA single-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1			Zm		0	0		Rv	1	0	1			Zn		0	0	0		off3	
																U															

SDOT ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, { <Zm1>.<Tb>-<Zm2>.<Tb> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1			Zm		0	1	0		Rv	1	0	1			Zn		0	0	0	0	off3
																U															

SDOT ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, { <Zm1>.<Tb>-<Zm4>.<Tb> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Tb> Is the size specifier, encoded in "sz":
- | sz | <Tb> |
|----|------|
| 0  | B    |
| 1  | H    |
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  
For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SEL

Multi-vector conditionally select elements from two vectors

Read active elements from the two or four first source vectors and inactive elements from the two or four second source vectors and place in the corresponding elements of the two or four destination vectors.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1		Zm	0	1	0	0	PNg		Zn	0		Zd	0									

SEL { <Zd1>.<T>-<Zd2>.<T> }, <PNg>, { <Zn1>.<T>-<Zn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer d = UInt(Zd:'0');
integer g = UInt('1':PNg);
constant integer nreg = 2;
```

Four registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1		Zm	0	1	1	0	0	PNg		Zn	0	0		Zd	0	0						

SEL { <Zd1>.<T>-<Zd4>.<T> }, <PNg>, { <Zn1>.<T>-<Zn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer d = UInt(Zd:'00');
integer g = UInt('1':PNg);
constant integer nreg = 4;
```

Assembler Symbols

- <Zd1>

For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.  
  
For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D
- <Zd4>

Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2>

Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Zn1>	For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm1>	For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
<Zm4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
<Zm2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            Elem[results[r], e, esize] = Elem[operand1, e, esize];
        else
            Elem[results[r], e, esize] = Elem[operand2, e, esize];

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMAX (multiple and single vector)

Multi-vector signed maximum by vector

Determine the signed maximum of elements of the second source vector and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0		Zm		1	0	1	0	0	0	0	0	0	0	0	0	0	Zdn				0
U																															

SMAX { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt('0':Zm);
constant integer nreg = 2;
boolean unsigned = FALSE;
```

Four registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0		Zm		1	0	1	0	1	0	1	0	0	0	0	0	0	Zdn		0	0	0
U																															

SMAX { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt('0':Zm);
constant integer nreg = 4;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        integer element1 = Int(Elem[operand1, e, esize], unsigned);
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer res = Max(element1, element2);
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMAX (multiple vectors)

Multi-vector signed maximum

Determine the signed maximum of elements of the two or four second source vectors and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	Zm			0	1	0	1	1	0	0	0			0	0	0	0	Zdn			0	
U																															

SMAX { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt(Zm:'0');
constant integer nreg = 2;
boolean unsigned = FALSE;
```

Four registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	Zm			0	0	1	0	1	1	1	0	0			0	0	0	0	Zdn		0	0
U																															

SMAX { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt(Zm:'00');
constant integer nreg = 4;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
<Zm1>	For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
<Zm4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
<Zm2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        integer element1 = Int(Elem[operand1, e, esize], unsigned);
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer res = Max(element1, element2);
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SMIN (multiple and single vector)

Multi-vector signed minimum by vector

Determine the signed minimum of elements of the second source vector and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	0	0	0	0	0	0	0	1	Zdn				0
U																																

SMIN { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt('0':Zm);
constant integer nreg = 2;
boolean unsigned = FALSE;
```

Four registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	1	0	0	0	0	0	1	Zdn				0	0
U																																

SMIN { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt('0':Zm);
constant integer nreg = 4;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        integer element1 = Int(Elem[operand1, e, esize], unsigned);
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer res = Min(element1, element2);
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMIN (multiple vectors)

Multi-vector signed minimum

Determine the signed minimum of elements of the two or four second source vectors and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	Zm			0	1	0	1	1	0	0	0			0	0	0	1	Zdn			0	
U																															

SMIN { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt(Zm:'0');
constant integer nreg = 2;
boolean unsigned = FALSE;
```

Four registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	size	1	Zm			0	0	1	0	1	1	1	0	0			0	0	0	1	Zdn			0	0
U																																

SMIN { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt(Zm:'00');
constant integer nreg = 4;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
<Zm1>	For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
<Zm4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
<Zm2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        integer element1 = Int(Elem[operand1, e, esize], unsigned);
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer res = Min(element1, element2);
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SMLAL (multiple and indexed vector)

Multi-vector signed integer multiply-add long by indexed element

This signed integer multiply-add long instruction multiplies each signed 16-bit element in the one, two, or four first source vectors with each signed 16-bit indexed element of the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the ZA double-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to 7, encoded in 3 bits. The lowest of the two consecutive vector numbers forming the double-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: [One ZA double-vector](#) , [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

### One ZA double-vector

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	0	0	0	0	1	1	1	0	0	Zm				i3h	Rv		1	i3l		Zn				0	0	off3					
																												U		S			

**SMLAL** ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 1;
```

### Two ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	0	0	0	0	1	1	1	0	1	Zm				0	Rv		1	i3h		Zn				0	0	0	i3l	off2			
																												U		S			

**SMLAL** ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 2;
```

### Four ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
1	1	0	0	0	0	0	1	1	1	0	1	Zm				1	Rv		1	i3h		Zn				0	0	0	0	i3l	off2								
																												U				S							

```
SMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
<offs2>	For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 1
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer segmentbase = e - (e MOD eltspersegment);
      integer s = 2 * segmentbase + index;
      integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
      integer element2 = SInt(Elem[operand2, s, esize DIV 2]);
      bits(esize) product = (element1 * element2)<esize-1:0>;
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SMLAL (multiple and single vector)

Multi-vector signed integer multiply-add long by vector

This signed integer multiply-add long instruction multiplies each signed 16-bit element in the one, two, or four first source vectors with each signed 16-bit element in the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated.

It has encodings from 3 classes: [One ZA double-vector](#) , [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

### One ZA double-vector

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm			0	Rv		0	1	1	Zn			0	0	off3					
																								U		S					

SMLAL ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
constant integer nreg = 1;
```

### Two ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm			0	Rv		0	1	0	Zn			0	0	0	off2				
																								U		S					

SMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
constant integer nreg = 2;
```

### Four ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	1	Zm			0	Rv		0	1	0	Zn			0	0	0	off2				
																								U		S					



```
SMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
<offs2>	For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[(n+r) MOD 32, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 1
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
      integer element2 = SInt(Elem[operand2, 2 * e + i, esize DIV 2]);
      bits(esize) product = (element1 * element2)<esize-1:0>;
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SMLAL (multiple vectors)

Multi-vector signed integer multiply-add long

This signed integer multiply-add long instruction multiplies each signed 16-bit element in the two or four first source vectors with each signed 16-bit element in the two or four second source vectors, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

### Two ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	Zm				0	0	Rv		0	1	0	Zn				0	0	0	0	off2	
																								U				S			

SMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off2:'0');
constant integer nreg = 2;
```

### Four ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	1	1	Zm				0	1	0	Rv		0	1	0	Zn			0	0	0	0	0	off2	
																								U				S				

SMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off2:'0');
constant integer nreg = 4;
```

### Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs1> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
- <offs2> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
- <Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm1>	For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
<Zm4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
<Zm2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = SInt(Elem[operand2, 2 * e + i, esize DIV 2]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SMLALL (multiple and indexed vector)

Multi-vector signed integer multiply-add long-long by indexed element

This signed integer multiply-add long-long instruction multiplies each signed 8-bit or 16-bit element in the one, two, or four first source vectors with each signed 8-bit or 16-bit indexed element of second source vector, widens each product to 32-bits or 64-bits and destructively adds these values to the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 3 to 4 bits depending on the size of the element. The lowest of the four consecutive vector numbers forming the quad-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 6 classes: [One ZA quad-vector of 32-bit elements](#) , [One ZA quad-vector of 64-bit elements](#) , [Two ZA quad-vectors of 32-bit elements](#) , [Two ZA quad-vectors of 64-bit elements](#) , [Four ZA quad-vectors of 32-bit elements](#) and [Four ZA quad-vectors of 64-bit elements](#)

## One ZA quad-vector of 32-bit elements

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	0	0	0	0	Zm			i4h		Rv		i4l			Zn			0		0		0		off2	
U S																																

```
SMLALL ZA.S[<Wv>, <offs1>:<offs4>], <Zn>.B, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 1;
```

## One ZA quad-vector of 64-bit elements

(FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	0	Zm			i3h	Rv	0	i3l	Zn			0	0	0	off2						
U S																															

SMLALL ZA.D[<Wv>, <offs1>:<offs4>], <Zn>.H, <Zm>.H[<index>]

```

if !(HaveSME() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i3h:i3l);
constant integer nreq = 1;

```

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm			0	Rv	0	i4h	Zn			0	0	0	i4l	o1					
U S																															

SMLALL ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 2;
```

## (FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	0	0	0	0	1	1	0	0	1	Zm			0	Rv	0	0	i3h	Zn			0	0	0	i3l	o1						
																												U		S			

SMLALL ZA.D[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 2;
```

**(FEAT\_SME2)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	0	0	0	0	1	0	0	0	1	Zm			1	Rv	0	i4h	Zn			0	0	0	0	i4l	o1						
																												U		S			

```
SMLALL ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 4;
```

(FEAT SME I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
1	1	0	0	0	0	0	1	1	0	0	1	Zm			1	Rv	0	0	i3h	Zn			0	0	0	0	i3l	o1									
																												U S									

```
SMLALL ZA.D[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA quad-vector of 32-bit elements and one ZA quad-vector of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.  For the four ZA quad-vectors of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.
<offs4>	For the one ZA quad-vector of 32-bit elements and one ZA quad-vector of 64-bit elements variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "off2" field times 4 plus 3.  For the four ZA quad-vectors of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	For the two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA quad-vectors of 32-bit elements and four ZA quad-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	For the four ZA quad-vectors of 32-bit elements, one ZA quad-vector of 32-bit elements and two ZA quad-vectors of 32-bit elements variant: is the element index, in the range 0 to 15, encoded in the "i4h:i4l" fields.  For the four ZA quad-vectors of 64-bit elements, one ZA quad-vector of 64-bit elements and two ZA quad-vectors of 64-bit elements variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 3
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer segmentbase = e - (e MOD eltspersegment);
      integer s = 4 * segmentbase + index;
      integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
      integer element2 = SInt(Elem[operand2, s, esize DIV 4]);
      bits(esize) product = (element1 * element2) < esize-1:0 >;
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## SMLALL (multiple and single vector)

Multi-vector signed integer multiply-add long-long by vector

This signed integer multiply-add long-long instruction multiplies each signed 8-bit or 16-bit element in the one, two, or four first source vectors with each signed 8-bit or 16-bit element in the second source vector, widens each product to 32-bits or 64-bits and destructively adds these values to the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 3 classes: [One ZA quad-vector](#) , [Two ZA quad-vectors](#) and [Four ZA quad-vectors](#)

### One ZA quad-vector

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	0	sz	1	0	Zm			0		Rv		0			0	1	Zn				0		0	0	off2
																												U		S		

SMLALL ZA.<T>[<Wv>, <offs1>:<offs4>], <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
constant integer nreg = 1;
```

### Two ZA quad-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	0	sz	1	0	Zm			0	Rv		0	0	0	Zn				0	0	0	0	0	0	0	o1
																												U	S			

SMLALL ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 2;
```

### Four ZA quad-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	0	sz	1	1	Zm			0		Rv		0			0	0	Zn			0		0	0	0	o1
																												U		S		

SMLALL ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs1> For the one ZA quad-vector variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.  
For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offs4> For the one ZA quad-vector variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "off2" field times 4 plus 3.  
For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Tb> Is the size specifier, encoded in “sz”:

sz	<Tb>
0	B
1	H

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            bits(esize) product = (element1 * element2) < esize-1:0 >;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLALL (multiple vectors)

Multi-vector signed integer multiply-add long-long

This signed integer multiply-add long-long instruction multiplies each signed 8-bit or 16-bit element in the two or four first source vectors with each signed 8-bit or 16-bit element in the one, two, or four second source vectors, widens each product to 32-bits or 64-bits and destructively adds these values to the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: [Two ZA quad-vectors](#) and [Four ZA quad-vectors](#)

Two ZA quad-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	sz	1			Zm		0	0		Rv	0	0	0			Zn		0	0	0	0	0	0	o1
																												U		S		

SMLALL ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, { <Zm1>.<Tb>-<Zm2>.<Tb> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(o1:'00');
constant integer nreg = 2;
```

Four ZA quad-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1		Zm		0	1	0		Rv	0	0	0		Zn		0	0	0	0	0	0	o1
																												U		S	

SMLALL ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, { <Zm1>.<Tb>-<Zm4>.<Tb> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>	Is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.						
<offs4>	Is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.						
<Zn1>	For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.						
<Tb>	Is the size specifier, encoded in "sz": <table border="1"> <thead> <tr> <th>sz</th><th>&lt;Tb&gt;</th></tr> </thead> <tbody> <tr> <td>0</td><td>B</td></tr> <tr> <td>1</td><td>H</td></tr> </tbody> </table>	sz	<Tb>	0	B	1	H
sz	<Tb>						
0	B						
1	H						
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.						
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.						
<Zm1>	For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.						
<Zm4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.						
<Zm2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.						

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m+r, VL];
  for i = 0 to 3
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
      integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
      bits(esize) product = (element1 * element2)<esize-1:0>;
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



## SMLS (multiple and indexed vector)

Multi-vector signed integer multiply-subtract long by indexed element

This signed integer multiply-subtract long instruction multiplies each signed 16-bit element in the one, two, or four first source vectors with each signed 16-bit indexed element of the second source vector, widens each product to 32-bits and destructively subtracts these values from the corresponding 32-bit elements of the ZA double-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to 7, encoded in 3 bits. The lowest of the two consecutive vector numbers forming the double-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: [One ZA double-vector](#) , [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

### One ZA double-vector

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	0	Zm			i3h	Rv	1	i3l	Zn			0	1	off3							
																												U		S	

**SMLS** ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 1;
```

### Two ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm			0	Rv	1	i3h	Zn			0	0	1	i3l	off2					
																												U		S	

**SMLS** ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 2;
```

### Four ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm			1	Rv	1	i3h	Zn			0	0	0	1	i3l	off2				
																												U		S	

```
SMLS_L ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
<offs2>	For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer segmentbase = e - (e MOD eltspersegment);
            integer s = 2 * segmentbase + index;
            integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = SInt(Elem[operand2, s, esize DIV 2]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```



## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# SMLSL (multiple and single vector)

Multi-vector signed integer multiply-subtract long by vector

This signed integer multiply-subtract long instruction multiplies each signed 16-bit element in the one, two, or four first source vectors with each signed 16-bit element in the second source vector, widens each product to 32-bits and destructively subtracts these values from the corresponding 32-bit elements of the ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated.

It has encodings from 3 classes: [One ZA double-vector](#) , [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

## One ZA double-vector

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm			0	Rv		0	1	1	Zn			0	1	off3					
																								U		S					

SMLSL ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
constant integer nreg = 1;
```

## Two ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm			0	Rv		0	1	0	Zn			0	1	0	off2				
																								U		S					

SMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
constant integer nreg = 2;
```

## Four ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	1	Zm			0	Rv		0	1	0	Zn			0	1	0	off2				
																								U		S					

```
SMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
<offs2>	For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[(n+r) MOD 32, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 1
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
      integer element2 = SInt(Elem[operand2, 2 * e + i, esize DIV 2]);
      bits(esize) product = (element1 * element2)<esize-1:0>;
      Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SMLS (multiple vectors)

Multi-vector signed integer multiply-subtract long

This signed integer multiply-subtract long instruction multiplies each signed 16-bit element in the two or four first source vectors with each signed 16-bit element in the two or four second source vectors, widens each product to 32-bits and destructively subtracts these values from the corresponding 32-bit elements of the ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

### Two ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	Zm				0	0	Rv		0	1	0	Zn				0	0	1	0	off2	
																								U				S			

SMLS ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off2:'0');
constant integer nreg = 2;
```

### Four ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	1	1	Zm				0	1	0	Rv		0	1	0	Zn			0	0	0	1	0	off2	
																								U				S				

SMLS ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off2:'0');
constant integer nreg = 4;
```

### Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs1> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
- <offs2> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
- <Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm1>	For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
<Zm4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
<Zm2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = SInt(Elem[operand2, 2 * e + i, esize DIV 2]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# SMLSLL (multiple and indexed vector)

Multi-vector signed integer multiply-subtract long-long by indexed element

This signed integer multiply-subtract long-long instruction multiplies each signed 8-bit or 16-bit element in the one, two, or four first source vectors with each signed 8-bit or 16-bit indexed element of second source vector, widens each product to 32-bits or 64-bits and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 3 to 4 bits depending on the size of the element. The lowest of the four consecutive vector numbers forming the quad-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

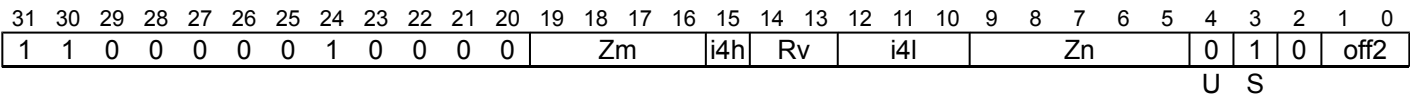
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 6 classes: [One ZA quad-vector of 32-bit elements](#) , [One ZA quad-vector of 64-bit elements](#) , [Two ZA quad-vectors of 32-bit elements](#) , [Two ZA quad-vectors of 64-bit elements](#) , [Four ZA quad-vectors of 32-bit elements](#) and [Four ZA quad-vectors of 64-bit elements](#)

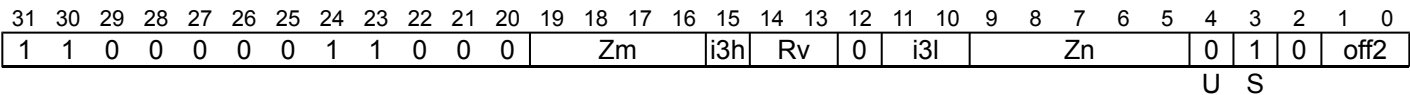
## One ZA quad-vector of 32-bit elements (FEAT\_SME2)



SMLSLL ZA.S[<Wv>, <offs1>:<offs4>], <Zn>.B, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 1;
```

## One ZA quad-vector of 64-bit elements (FEAT\_SME\_I16I64)



SMLSLL ZA.D[<Wv>, <offs1>:<offs4>], <Zn>.H, <Zm>.H[<index>]

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 1;
```

## Two ZA quad-vectors of 32-bit elements (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm			0	Rv	0	i4h	Zn			0	0	1	i4l	o1					
																										U	S				

**SMLSLL** ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 2;
```

## Two ZA quad-vectors of 64-bit elements (FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm			0	Rv	0	0	i3h	Zn			0	0	1	i3l	o1				
																										U	S				

**SMLSLL** ZA.D[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 2;
```

## Four ZA quad-vectors of 32-bit elements (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm			1	Rv	0	i4h	Zn			0	0	0	0	1	i4l	o1			
																										U	S				

**SMLSLL** ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 4;
```

## Four ZA quad-vectors of 64-bit elements (FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm			1	Rv	0	0	i3h	Zn			0	0	0	1	i3l	o1			
																										U	S				



SMLSLL ZA.D[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA quad-vector of 32-bit elements and one ZA quad-vector of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.  For the four ZA quad-vectors of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.
<offs4>	For the one ZA quad-vector of 32-bit elements and one ZA quad-vector of 64-bit elements variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "off2" field times 4 plus 3.  For the four ZA quad-vectors of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	For the two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA quad-vectors of 32-bit elements and four ZA quad-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	For the four ZA quad-vectors of 32-bit elements, one ZA quad-vector of 32-bit elements and two ZA quad-vectors of 32-bit elements variant: is the element index, in the range 0 to 15, encoded in the "i4h:i4l" fields.  For the four ZA quad-vectors of 64-bit elements, one ZA quad-vector of 64-bit elements and two ZA quad-vectors of 64-bit elements variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 3
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer segmentbase = e - (e MOD eltspersegment);
      integer s = 4 * segmentbase + index;
      integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
      integer element2 = SInt(Elem[operand2, s, esize DIV 4]);
      bits(esize) product = (element1 * element2)<esize-1:0>;
      Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SMLSLL (multiple and single vector)

Multi-vector signed integer multiply-subtract long-long by vector

This signed integer multiply-subtract long-long instruction multiplies each signed 8-bit or 16-bit element in the one, two, or four first source vectors with each signed 8-bit or 16-bit element in the second source vector, widens each product to 32-bits or 64-bits and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 3 classes: [One ZA quad-vector](#) , [Two ZA quad-vectors](#) and [Four ZA quad-vectors](#)

### One ZA quad-vector

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm			0	Rv		0	0	1	Zn				0	1	0	off2			
																												U		S	

SMLSLL ZA.<T>[<Wv>, <offs1>:<offs4>], <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
constant integer nreg = 1;
```

### Two ZA quad-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	0	0	0	0	1	0	sz	1	0	Zm			0		Rv		0			0	0	Zn				0		1	0	0	o1
																														U		S	

SMLSLL ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 2;
```

### Four ZA quad-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1		Zm		0		Rv		0	0	0			Zn		0	1	0	0	o1	
																														U	S

SMLSLL ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs1> For the one ZA quad-vector variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.  
For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offs4> For the one ZA quad-vector variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "off2" field times 4 plus 3.  
For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Tb> Is the size specifier, encoded in “sz”:

sz	<Tb>
0	B
1	H

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            bits(esize) product = (element1 * element2) < esize-1:0 >;
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLSLL (multiple vectors)

Multi-vector signed integer multiply-subtract long-long

This signed integer multiply-subtract long-long instruction multiplies each signed 8-bit or 16-bit element in the two or four first source vectors with each signed 8-bit or 16-bit element in the one, two, or four second source vectors, widens each product to 32-bits or 64-bits and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: [Two ZA quad-vectors](#) and [Four ZA quad-vectors](#)

Two ZA quad-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	sz	1			Zm		0	0		Rv	0	0	0			Zn		0	0	1	0	0	o1	
																														U		S

SMLSLL ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, { <Zm1>.<Tb>-<Zm2>.<Tb> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(o1:'00');
constant integer nreg = 2;
```

Four ZA quad-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	sz	1		Zm		0	1	0		Rv	0	0	0			Zn		0	0	0	1	0	0	o1
																														U		S

SMLSLL ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, { <Zm1>.<Tb>-<Zm4>.<Tb> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

- <offs1> Is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.
- <offs4> Is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.
- <Zn1> For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	B
1	H

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  
For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m+r, VL];
  for i = 0 to 3
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
      integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
      bits(esize) product = (element1 * element2)<esize-1:0>;
      Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.





SMOPA (2-way)

Signed integer sum of outer products and accumulate

This instruction works with a 32-bit element ZA tile.

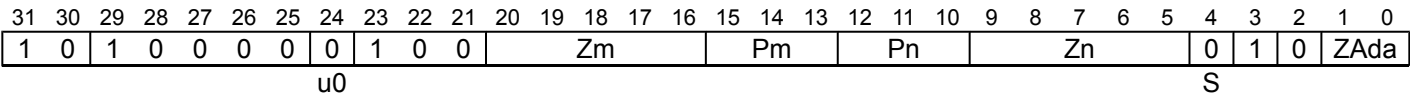
The signed integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. The first source holds SVLS×2 sub-matrix of signed 16-bit integer values, and the second source holds 2×SVLS sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is inactive, it is treated as having the value 0.

The resulting SVLS×SVLS widened 32-bit integer sum of outer products is then destructively added to the 32-bit integer destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a SVLS×2 sub-matrix, and each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a 2×SVLS sub-matrix.

SME2  
(FEAT\_SME2)



SMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean unsigned = FALSE;
```

Assembler Symbols

- <ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) sum = Elem[operand3, row*dim+col, esize];
    for k = 0 to 1
      if ActivePredicateElement(mask1, 2*row + k, esize DIV 2) &&
         ActivePredicateElement(mask2, 2*col + k, esize DIV 2) then
        prod = (Int(Elem[operand1, 2*row + k, esize DIV 2], unsigned) *
               Int(Elem[operand2, 2*col + k, esize DIV 2], unsigned));
        if sub_op then prod = -prod;
        sum = sum + prod;

    Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMOPA (4-way)

Signed integer sum of outer products and accumulate

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The signed integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds  $SVL_S \times 4$  sub-matrix of signed 8-bit integer values, and the second source holds  $4 \times SVL_S$  sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds  $SVL_D \times 4$  sub-matrix of signed 16-bit integer values, and the second source holds  $4 \times SVL_D$  sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting  $SVL_S \times SVL_S$  widened 32-bit integer or  $SVL_D \times SVL_D$  widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a  $SVL_S \times 4$  sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a  $4 \times SVL_S$  sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a  $SVL_D \times 4$  sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a  $4 \times SVL_D$  sub-matrix.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

32-bit  
(FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	1	0	0	Zm				Pm				Pn				Zn				0	0	0	ZAda	
u0								u1				S																			

SMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;
```

64-bit  
(FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	0	1	0	0	0	0	0	1	1	0	Zm				Pm				Pn				Zn				0	0	ZAda				
u0											u1				S																		

SMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;
```

## Assembler Symbols

- <ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.  
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) sum = Elem[operand3, row*dim+col, esize];
    for k = 0 to 3
      if ActivePredicateElement(mask1, 4*row + k, esize DIV 4) &&
         ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
        prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
               Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
        if sub_op then prod = -prod;
        sum = sum + prod;

    Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.



SMOPS (2-way)

Signed integer sum of outer products and subtract

This instruction works with a 32-bit element ZA tile.

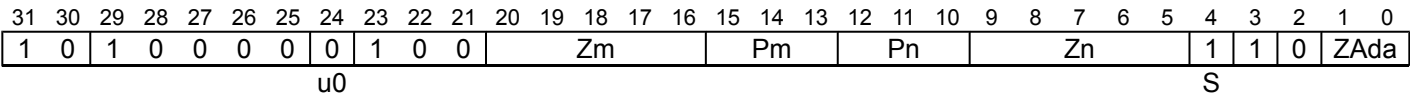
The signed integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. The first source holds  $SVL_S \times 2$  sub-matrix of signed 16-bit integer values, and the second source holds  $2 \times SVL_S$  sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is inactive, it is treated as having the value 0.

The resulting  $SVL_S \times SVL_S$  widened 32-bit integer sum of outer products is then destructively subtracted from the 32-bit integer destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a  $SVL_S \times 2$  sub-matrix, and each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a  $2 \times SVL_S$  sub-matrix.

SME2  
(FEAT\_SME2)



SMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean unsigned = FALSE;
```

Assembler Symbols

- <ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) sum = Elem[operand3, row*dim+col, esize];
    for k = 0 to 1
      if ActivePredicateElement(mask1, 2*row + k, esize DIV 2) &&
         ActivePredicateElement(mask2, 2*col + k, esize DIV 2) then
        prod = (Int(Elem[operand1, 2*row + k, esize DIV 2], unsigned) *
               Int(Elem[operand2, 2*col + k, esize DIV 2], unsigned));
        if sub_op then prod = -prod;
        sum = sum + prod;

    Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMOPS (4-way)

Signed integer sum of outer products and subtract

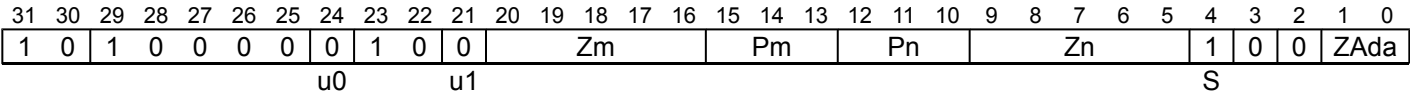
The 8-bit integer variant works with a 32-bit element ZA tile.  
The 16-bit integer variant works with a 64-bit element ZA tile.  
The signed integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds  $SVL_S \times 4$  sub-matrix of signed 8-bit integer values, and the second source holds  $4 \times SVL_S$  sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds  $SVL_D \times 4$  sub-matrix of signed 16-bit integer values, and the second source holds  $4 \times SVL_D$  sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.  
The resulting  $SVL_S \times SVL_S$  widened 32-bit integer or  $SVL_D \times SVL_D$  widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a  $SVL_S \times 4$  sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a  $4 \times SVL_S$  sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a  $SVL_D \times 4$  sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a  $4 \times SVL_D$  sub-matrix.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.  
It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

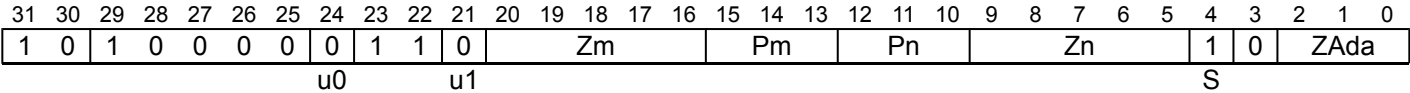
32-bit  
(FEAT\_SME)



SMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;
```

64-bit  
(FEAT\_SME\_I16I64)





SMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;
```

## Assembler Symbols

<ZAda>	For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) sum = Elem[operand3, row*dim+col, esize];
    for k = 0 to 3
      if ActivePredicateElement(mask1, 4*row + k, esize DIV 4) &&
         ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
        prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
               Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
        if sub_op then prod = -prod;
        sum = sum + prod;

    Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

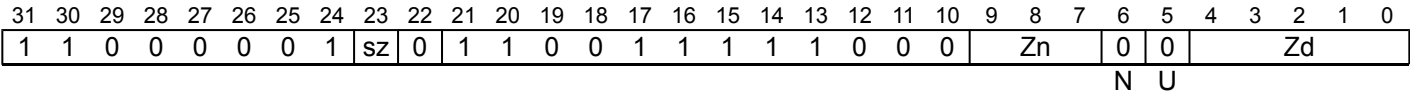


SQCVT (four registers)

Multi-vector signed saturating extract narrow

Saturate the signed integer value in each element of the four source vectors to quarter the original source element width, and place the results in the quarter-width destination elements.  
This instruction is unpredicated.

SME2  
(FEAT\_SME2)



SQCVT <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(sz);
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <T> Is the size specifier, encoded in "sz":

sz	<T>
0	B
1	H
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	S
1	D
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
bits(VL) result;

for r = 0 to 3
  bits(VL) operand = Z[n+r, VL];
  for e = 0 to elements-1
    integer element = SInt(Elem[operand, e, 4 * esize]);
    Elem[result, r*elements + e, esize] = SignedSat(element, esize);

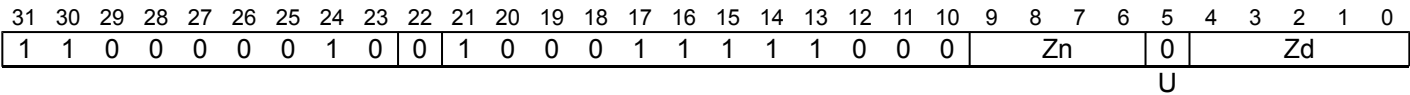
Z[d, VL] = result;
```

SQCVT (two registers)

Multi-vector signed saturating extract narrow

Saturate the signed integer value in each element of the two source vectors to half the original source element width, and place the results in the half-width destination elements.  
This instruction is unpredicated.

SME2  
(FEAT\_SME2)



```
SQCVT <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;

for r = 0 to 1
  bits(VL) operand = Z[n+r, VL];
  for e = 0 to elements-1
    integer element = SInt(Elem[operand, e, 2 * esize]);
    Elem[result, r*elements + e, esize] = SignedSat(element, esize);

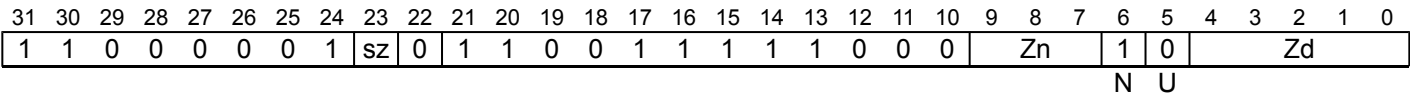
Z[d, VL] = result;
```

SQCVTN

Multi-vector signed saturating extract narrow and interleave

Saturate the signed integer value in each element of the four source vectors to quarter the original source element width, and place the four-way interleaved results in the quarter-width destination elements.  
This instruction is unpredicated.

SME2  
(FEAT\_SME2)



SQCVTN <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(sz);
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <T> Is the size specifier, encoded in "sz":

sz	<T>
0	B
1	H
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	S
1	D
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
bits(VL) result;

for e = 0 to elements-1
  for i = 0 to 3
    bits(VL) operand = Z[n+i, VL];
    integer element = SInt(Elem[operand, e, 4 * esize]);
    Elem[result, 4*e + i, esize] = SignedSat(element, esize);

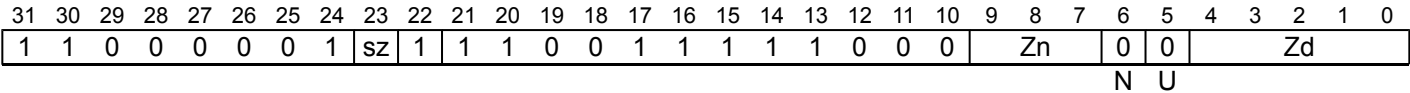
Z[d, VL] = result;
```

SQCVTU (four registers)

Multi-vector signed saturating unsigned extract narrow

Saturate the signed integer value in each element of the four source vectors to unsigned integer value that is quarter the original source element width, and place the results in the quarter-width destination elements.  
This instruction is unpredicated.

SME2  
(FEAT\_SME2)



```
SQCVTU <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(sz);
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <T> Is the size specifier, encoded in "sz":

sz	<T>
0	B
1	H
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	S
1	D
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
bits(VL) result;

for r = 0 to 3
  bits(VL) operand = Z[n+r, VL];
  for e = 0 to elements-1
    integer element = SInt(Elem[operand, e, 4 * esize]);
    Elem[result, r*elements + e, esize] = UnsignedSat(element, esize);

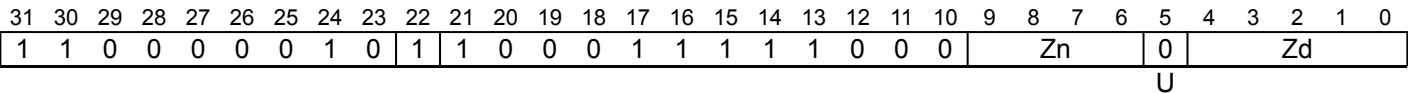
Z[d, VL] = result;
```

SQCVTU (two registers)

Multi-vector signed saturating unsigned extract narrow

Saturate the signed integer value in each element of the two source vectors to unsigned integer value that is half the original source element width, and place the results in the half-width destination elements.  
This instruction is unpredicated.

SME2  
(FEAT\_SME2)



```
SQCVTU <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;

for r = 0 to 1
  bits(VL) operand = Z[n+r, VL];
  for e = 0 to elements-1
    integer element = SInt(Elem[operand, e, 2 * esize]);
    Elem[result, r*elements + e, esize] = UnsignedSat(element, esize);

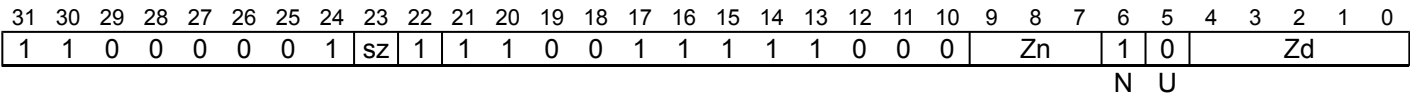
Z[d, VL] = result;
```

SQCVTUN

Multi-vector signed saturating unsigned extract narrow and interleave

Saturate the signed integer value in each element of the four source vectors to unsigned integer value that is quarter the original source element width, and place the four-way interleaved results in the quarter-width destination elements.  
This instruction is unpredicated.

SME2  
(FEAT\_SME2)



```
SQCVTUN <Zd>.<T>, { <Zn1>.<Tb>--<Zn4>.<Tb> }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(sz);
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <T> Is the size specifier, encoded in "sz":

sz	<T>
0	B
1	H
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	S
1	D
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
bits(VL) result;

for e = 0 to elements-1
  for i = 0 to 3
    bits(VL) operand = Z[n+i, VL];
    integer element = SInt(Elem[operand, e, 4 * esize]);
    Elem[result, 4*e + i, esize] = UnsignedSat(element, esize);

Z[d, VL] = result;
```



SQDMULH (multiple and single vector)

Multi-vector signed saturating doubling multiply high by vector

Multiply then double the corresponding signed elements of the two or four first source vectors and the signed elements of the second source vector, and destructively place the most significant half of the result in the corresponding elements of the two or four first source vectors. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ .

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0	Zm				1	0	1	0	0	1	0	0	0	0	Zdn				0		

SQDMULH { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt('0':Zm);
constant integer nreg = 2;
```

Four registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0	Zm				1	0	1	0	1	1	0	0	0	0	0	Zdn			0	0	

SQDMULH { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt('0':Zm);
constant integer nreg = 4;
```

Assembler Symbols

- <Zdn1>

For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D
- <Zdn4>

Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2>

Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        integer element1 = SInt(Elem[operand1, e, esize]);
        integer element2 = SInt(Elem[operand2, e, esize]);
        integer res = 2 * element1 * element2;
        Elem[results[r], e, esize] = SignedSat(res >> esize, esize);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SQDMULH (multiple vectors)

Multi-vector signed saturating doubling multiply high

Multiply then double the corresponding signed elements of the two or four first and second source vectors, and destructively place the most significant half of the result in the corresponding elements of the two or four first source vectors. Each result element is saturated to the N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ .

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1		Zm	0	1	0	1	1	0	1	0	0	0	0	0		Zdn						0

SQDMULH { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt(Zm:'0');
constant integer nreg = 2;
```

Four registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1		Zm	0	0	1	0	1	1	1	1	0	0	0	0	0		Zdn					0

SQDMULH { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt(Zm:'00');
constant integer nreg = 4;
```

Assembler Symbols

- <Zdn1>

For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.
- <T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D
- <Zdn4>

Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2>

Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm1>

For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        integer element1 = SInt(Elem[operand1, e, esize]);
        integer element2 = SInt(Elem[operand2, e, esize]);
        integer res = 2 * element1 * element2;
        Elem[results[r], e, esize] = SignedSat(res >> esize, esize);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

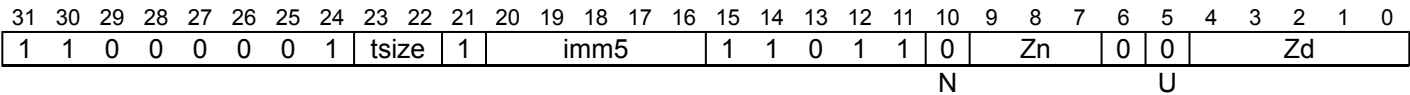
SQRSHR (four registers)

Multi-vector signed saturating rounding shift right narrow by immediate

Shift right by an immediate value, the signed integer value in each element of the four source vectors and place the rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2  
(FEAT\_SME2)



SQRSHR <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }, #<const>

```
if !HaveSME2() then UNDEFINED;
if tsize == '00' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
integer shift = (8 * esize) - UInt(tsize:imm5);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tsize":

tsize	<T>
00	RESERVED
01	B
1x	H

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "tsize":

tsize	<Tb>
00	RESERVED
01	S
1x	D

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<const> Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in "tsize:imm5".

## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
bits(VL) result;

for r = 0 to 3
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(4 * esize) element = Elem[operand, e, 4 * esize];
        integer res = (SInt(element) + (1 << (shift-1))) >> shift;
        Elem[result, r*elements + e, esize] = SignedSat(res, esize);

Z[d, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

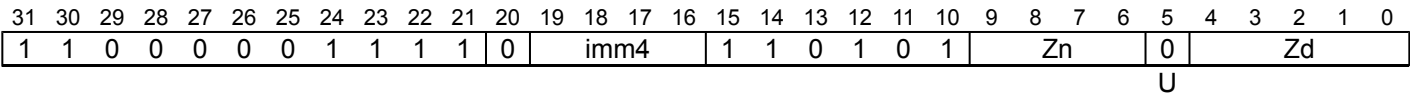
SQRSHR (two registers)

Multi-vector signed saturating rounding shift right narrow by immediate

Shift right by an immediate value, the signed integer value in each element of the two source vectors and place the rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SME2  
(FEAT\_SME2)



SQRSHR <Zd>.H, { <Zn1>.S-<Zn2>.S }, #<const>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
integer shift = esize - UInt(imm4);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <const> Is the immediate shift amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;

for r = 0 to 1
  bits(VL) operand = Z[n+r, VL];
  for e = 0 to elements-1
    bits(2 * esize) element = Elem[operand, e, 2 * esize];
    integer res = (SInt(element) + (1 << (shift-1))) >> shift;
    Elem[result, r*elements + e, esize] = SignedSat(res, esize);

Z[d, VL] = result;
```

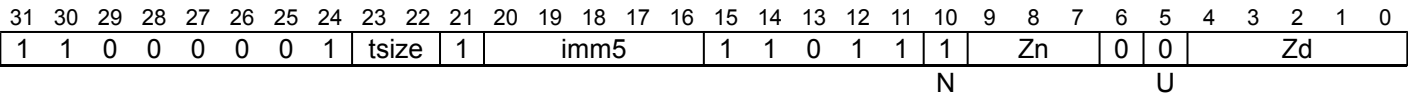
SQRSHRN

Multi-vector signed saturating rounding shift right narrow by immediate and interleave

Shift right by an immediate value, the signed integer value in each element of the four source vectors and place the four-way interleaved rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's signed integer range  $-2^{(N-1)}$  to  $(2^{(N-1)})-1$ . The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2  
(FEAT\_SME2)



```
SQRSHRN <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }, #<const>
```

```
if !HaveSME2() then UNDEFINED;
if tsize == '00' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
integer shift = (8 * esize) - UInt(tsize:imm5);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tsize":

tsize	<T>
00	RESERVED
01	B
1x	H

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "tsize":

tsize	<Tb>
00	RESERVED
01	S
1x	D

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<const> Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in "tsize:imm5".



## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
bits(VL) result;

for e = 0 to elements-1
    for i = 0 to 3
        bits(VL) operand = Z[n+i, VL];
        bits(4 * esize) element = Elem[operand, e, 4 * esize];
        integer res = (SInt(element) + (1 << (shift-1))) >> shift;
        Elem[result, 4*e + i, esize] = SignedSat(res, esize);

Z[d, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

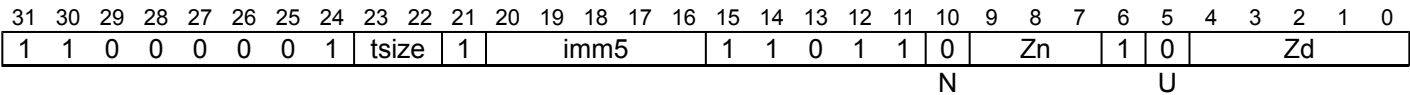
SQRSHRU (four registers)

Multi-vector signed saturating rounding shift right unsigned narrow by immediate

Shift right by an immediate value, the signed integer value in each element of the four source vectors and place the rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2  
(FEAT\_SME2)



SQRSHRU <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }, #<const>

```
if !HaveSME2() then UNDEFINED;
if tsize == '00' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
integer shift = (8 * esize) - UInt(tsize:imm5);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tsize":

tsize	<T>
00	RESERVED
01	B
1x	H

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "tsize":

tsize	<Tb>
00	RESERVED
01	S
1x	D

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<const> Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in "tsize:imm5".

## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
bits(VL) result;

for r = 0 to 3
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(4 * esize) element = Elem[operand, e, 4 * esize];
        integer res = (SInt(element) + (1 << (shift-1))) >> shift;
        Elem[result, r*elements + e, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

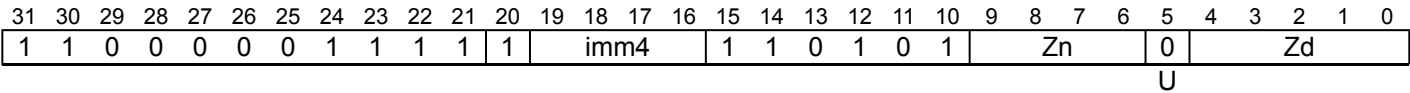
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SQRSHRU (two registers)

Multi-vector signed saturating rounding shift right unsigned narrow by immediate

Shift right by an immediate value, the signed integer value in each element of the two source vectors and place the rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to  $(2^N)-1$ . The immediate shift amount is an unsigned value in the range 1 to 16. This instruction is unpredicated.

### SME2 (FEAT\_SME2)



SQRSHRU <Zd>.H, { <Zn1>.S-<Zn2>.S }, #<const>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
integer shift = esize - UInt(imm4);
```

### Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <const> Is the immediate shift amount, in the range 1 to 16, encoded in the "imm4" field.

### Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;

for r = 0 to 1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(2 * esize) element = Elem[operand, e, 2 * esize];
        integer res = (SInt(element) + (1 << (shift-1))) >> shift;
        Elem[result, r*elements + e, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```

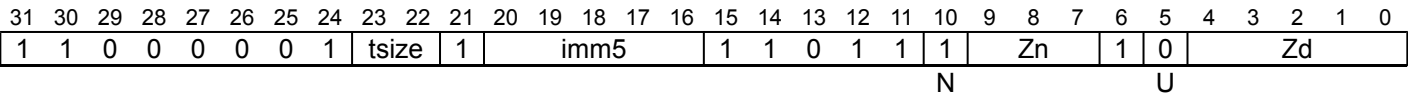
SQRSHRUN

Multi-vector signed saturating rounding shift right unsigned narrow by immediate and interleave

Shift right by an immediate value, the signed integer value in each element of the four source vectors and place the four-way interleaved rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2  
(FEAT\_SME2)



```
SQRSHRUN <Zd>.<T>, { <Zn1>.<Tb>--<Zn4>.<Tb> }, #<const>
```

```
if !HaveSME2() then UNDEFINED;
if tsize == '00' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
integer shift = (8 * esize) - UInt(tsize:imm5);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tsize":

tsize	<T>
00	RESERVED
01	B
1x	H

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "tsize":

tsize	<Tb>
00	RESERVED
01	S
1x	D

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<const> Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in "tsize:imm5".

## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
bits(VL) result;

for e = 0 to elements-1
    for i = 0 to 3
        bits(VL) operand = Z[n+i, VL];
        bits(4 * esize) element = Elem[operand, e, 4 * esize];
        integer res = (SInt(element) + (1 << (shift-1))) >> shift;
        Elem[result, 4*e + i, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SRSHL (multiple and single vector)

Multi-vector signed rounding shift left by vector

Shift the signed elements of the two or four first source vectors by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the two or four first source vectors. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	0	0	1	0	0		0	1	Zdn				0
U																																

SRSHL { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt('0':Zm);
constant integer nreg = 2;
```

Four registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	1	0	1	0	0	0	1	Zdn				0	0
U																																

SRSHL { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt('0':Zm);
constant integer nreg = 4;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.  
<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.  
<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        integer element = SInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
        integer res;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = (element + (1 << (shift - 1))) >> shift;
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SRSHL (multiple vectors)

Multi-vector signed rounding shift left

Shift the signed elements of the two or four first source vectors by corresponding elements of the two or four second source vectors and destructively place the rounded results in the corresponding elements of the two or four first source vectors. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1			Zm		0	1	0	1	1	0	0	1	0	0	0	1		Zdn				0
U																															

SRSHL { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt(Zm:'0');
constant integer nreg = 2;
```

Four registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1			Zm		0	0	1	0	1	1	1	0	1	0	0	0	1		Zdn		0	0
U																															

SRSHL { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt(Zm:'00');
constant integer nreg = 4;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.
- For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
  bits(VL) operand1 = Z[dn+r, VL];
  bits(VL) operand2 = Z[m+r, VL];
  for e = 0 to elements-1
    integer element = SInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer res;
    if shift >= 0 then
      res = element << shift;
    else
      shift = -shift;
      res = (element + (1 << (shift - 1))) >> shift;
    Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
  Z[dn+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# ST1B (scalar plus immediate, strided registers)

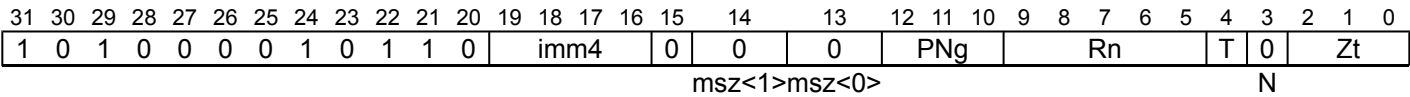
Contiguous store of bytes from multiple strided vectors (immediate index)

Contiguous store of bytes from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

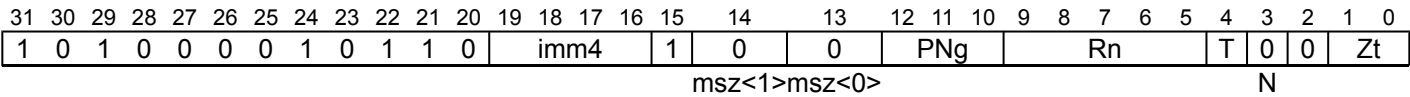
## Two registers (FEAT\_SME2)



ST1B { <Zt1>.B, <Zt2>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 8;
integer offset = SInt(imm4);
```

## Four registers (FEAT\_SME2)



ST1B { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 8;
integer offset = SInt(imm4);
```

## Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.  For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
        t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST1B (scalar plus scalar, strided registers)

Contiguous store of bytes from multiple strided vectors (scalar index)

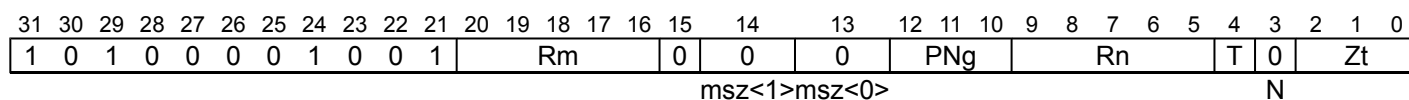
Contiguous store of bytes from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

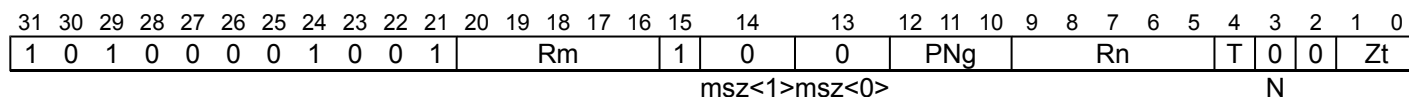


**ST1B** { <Zt1>.B, <Zt2>.B }, <PNg>, [<Xn|SP>, <Xm>]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 8;
```

### Four registers

(FEAT\_SME2)



**ST1B** { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>, [<Xn|SP>, <Xm>]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 8;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
        t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

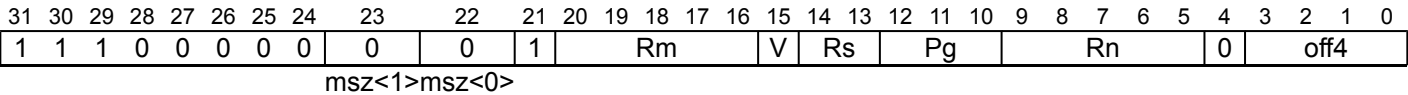
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST1B (scalar plus scalar, tile slice)

Contiguous store of bytes from 8-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 8-bit elements in a vector. The immediate offset is in the range 0 to 15. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is added to the base address. Inactive elements are not written to memory.

SME
(FEAT\_SME)



ST1B { ZA0<HV>.B[<Ws>, <offs>] }, <Pg>, [<Xn|SP>{, <Xm>}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = 0;
integer offset = UInt(off4);
constant integer esize = 8;
boolean vertical = V == '1';
```

Assembler Symbols

- <HV> Is the horizontal or vertical slice indicator, encoded in "V":
Table:
v | <HV>
0 | H
1 | V
- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs> Is the slice index offset, in the range 0 to 15, encoded in the "off4" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) moffs = X[m, 64];
bits(32) index = X[s, 32];
integer slice = (UInt(index) + offset) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
        ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    moffs = moffs + 1;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## ST1D (scalar plus immediate, strided registers)

Contiguous store of doublewords from multiple strided vectors (immediate index)

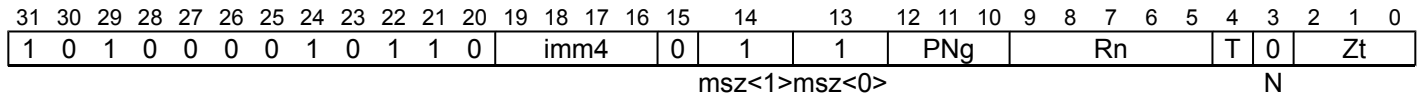
Contiguous store of doublewords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

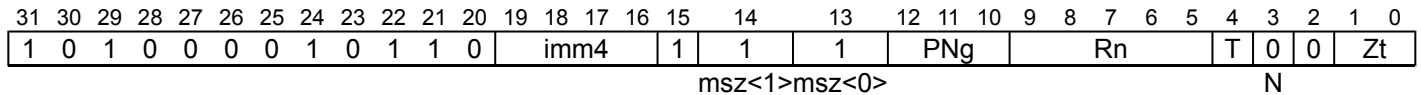


ST1D { <Zt1>.D, <Zt2>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SME2)



ST1D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.  For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
        t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST1D (scalar plus scalar, strided registers)

Contiguous store of doublewords from multiple strided vectors (scalar index)

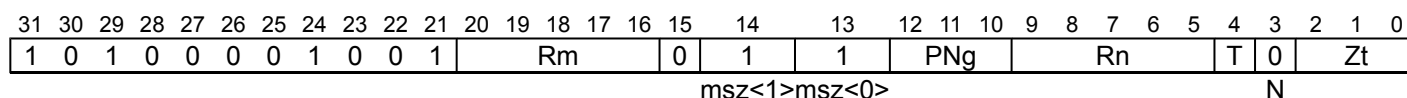
Contiguous store of doublewords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

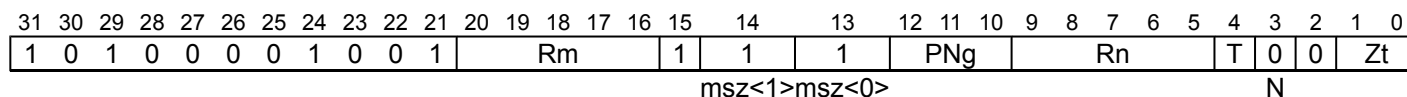


ST1D { <Zt1>.D, <Zt2>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 64;
```

### Four registers

(FEAT\_SME2)



ST1D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 64;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
        t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

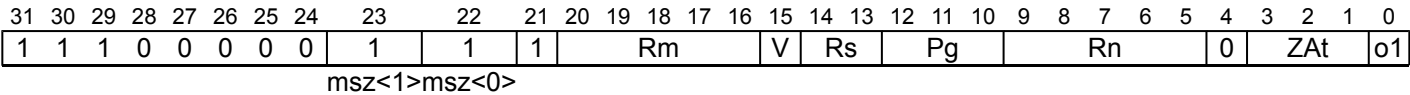
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST1D (scalar plus scalar, tile slice)

Contiguous store of doublewords from 64-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 64-bit elements in a vector. The immediate offset is in the range 0 to 1. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is multiplied by 8 and added to the base address. Inactive elements are not written to memory.

SME
(FEAT\_SME)



ST1D { <ZAt><HV>.D[<Ws>, <offs>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #3}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer offset = UInt(o1);
constant integer esize = 64;
boolean vertical = V == '1';
```

Assembler Symbols

- <ZAt> Is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAt" field.
- <HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V
- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs> Is the slice index offset, in the range 0 to 1, encoded in the "o1" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) moffs = X[m, 64];
bits(32) index = X[s, 32];
integer slice = (UInt(index) + offset) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
        ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    moffs = moffs + 1;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST1H (scalar plus immediate, strided registers)

Contiguous store of halfwords from multiple strided vectors (immediate index)

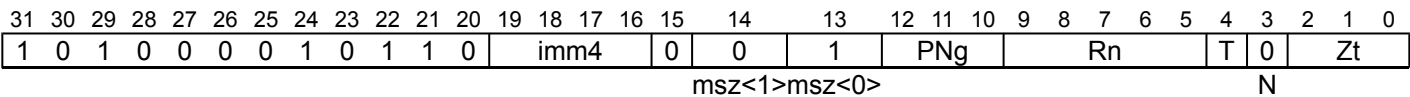
Contiguous store of halfwords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

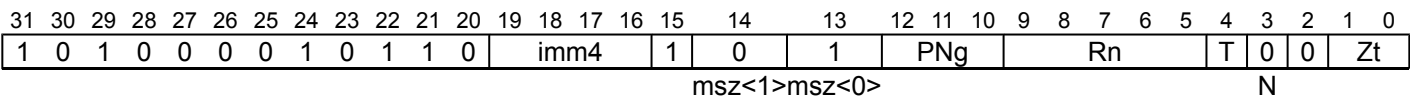


ST1H { <Zt1>.H, <Zt2>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 16;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SME2)



ST1H { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 16;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.  For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
        t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



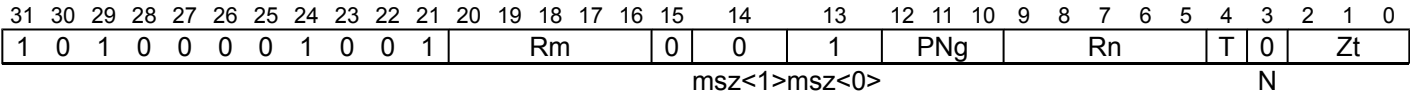
# ST1H (scalar plus scalar, strided registers)

Contiguous store of halfwords from multiple strided vectors (scalar index)

Contiguous store of halfwords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.  
Inactive elements are not written to memory.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

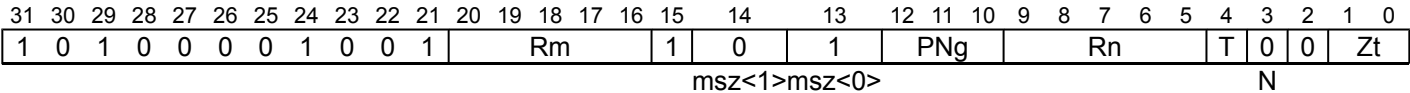
## Two registers (FEAT\_SME2)



ST1H { <Zt1>.H, <Zt2>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 16;
```

## Four registers (FEAT\_SME2)



ST1H { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 16;
```

## Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
        t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

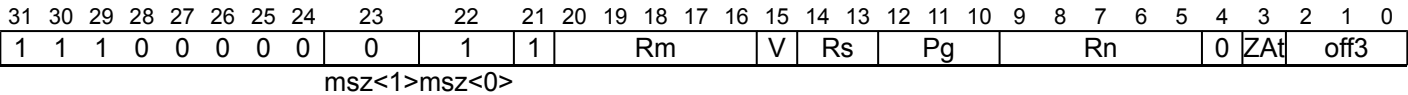
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST1H (scalar plus scalar, tile slice)

Contiguous store of halfwords from 16-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 16-bit elements in a vector. The immediate offset is in the range 0 to 7. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is multiplied by 2 and added to the base address. Inactive elements are not written to memory.

SME
(FEAT\_SME)



ST1H { <ZAt><HV>.H[<Ws>, <offs>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #1}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer offset = UInt(off3);
constant integer esize = 16;
boolean vertical = V == '1';
```

Assembler Symbols

- <ZAt> Is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAt" field.
- <HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V
- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs> Is the slice index offset, in the range 0 to 7, encoded in the "off3" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) moffs = X[m, 64];
bits(32) index = X[s, 32];
integer slice = (UInt(index) + offset) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
        ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    moffs = moffs + 1;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

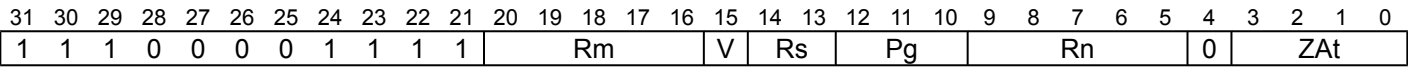
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST1Q

Contiguous store of quadwords from 128-bit element ZA tile slice

The slice number in the tile is selected by the slice index register, modulo the number of 128-bit elements in a Streaming SVE vector. The memory address is generated by scalar base and optional scalar offset which is multiplied by 16 and added to the base address. Inactive elements are not written to memory.

SME
(FEAT\_SME)



ST1Q { <ZAt><HV>.Q[<Ws>, <offs>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #4}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer offset = 0;
constant integer esize = 128;
boolean vertical = V == '1';
```

Assembler Symbols

- <ZAt> Is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAt" field.
- <HV> Is the horizontal or vertical slice indicator, encoded in "V":
Table:
V | <HV>
0 | H
1 | V
- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs> Is the slice index offset 0.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) moffs = X[m, 64];
bits(32) index = X[s, 32];
integer slice = (UInt(index) + offset) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
        ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    moffs = moffs + 1;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ST1W (scalar plus immediate, strided registers)

Contiguous store of words from multiple strided vectors (immediate index)

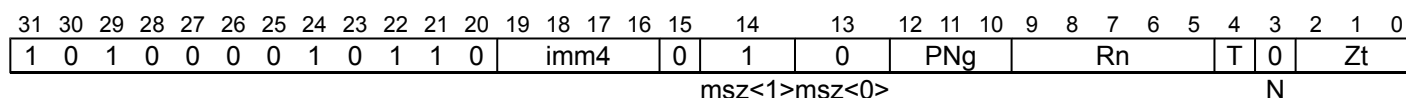
Contiguous store of words from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

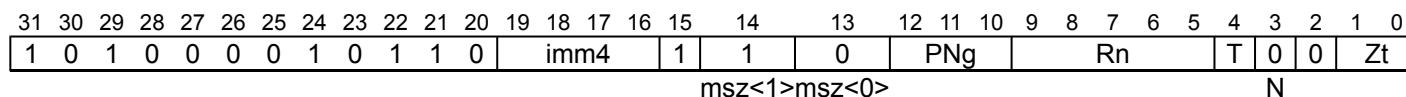


**ST1W** { <Zt1>.S, <Zt2>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 32;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SME2)



**ST1W** { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 32;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.  For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
        t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## ST1W (scalar plus scalar, strided registers)

Contiguous store of words from multiple strided vectors (scalar index)

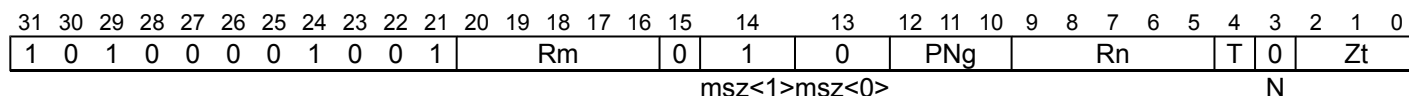
Contiguous store of words from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

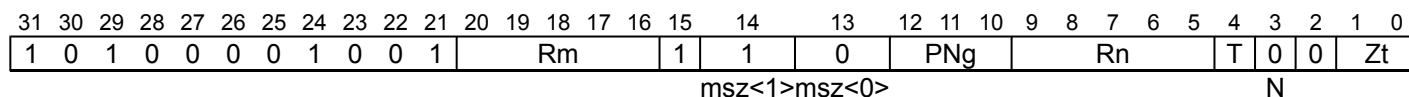


**ST1W** { <Zt1>.S, <Zt2>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 32;
```

### Four registers

(FEAT\_SME2)



**ST1W** { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 32;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
        t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

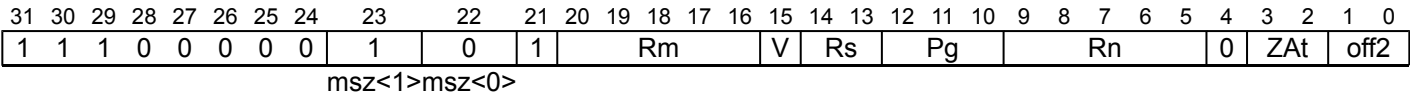
Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ST1W (scalar plus scalar, tile slice)

Contiguous store of words from 32-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 32-bit elements in a vector. The immediate offset is in the range 0 to 3. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is multiplied by 4 and added to the base address. Inactive elements are not written to memory.

SME
(FEAT\_SME)



ST1W { <ZAt><HV>.S[<Ws>, <offs>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #2}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer offset = UInt(off2);
constant integer esize = 32;
boolean vertical = V == '1';
```

Assembler Symbols

- <ZAt> Is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAt" field.
- <HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V
- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs> Is the slice index offset, in the range 0 to 3, encoded in the "off2" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) moffs = X[m, 64];
bits(32) index = X[s, 32];
integer slice = (UInt(index) + offset) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
        ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    moffs = moffs + 1;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1B (scalar plus immediate, strided registers)

Contiguous store non-temporal of bytes from multiple strided vectors (immediate index)

Contiguous store non-temporal of bytes from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

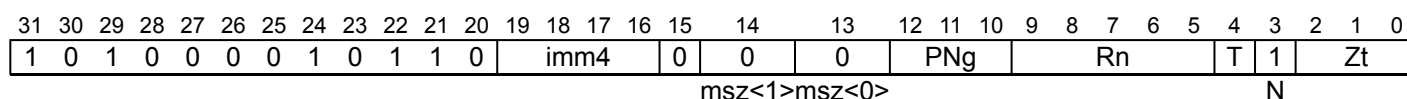
Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

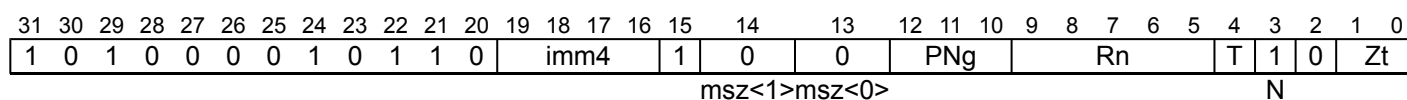


STNT1B { <Zt1>.B, <Zt2>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 8;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SME2)



STNT1B { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 8;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.  For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
        t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1B (scalar plus scalar, strided registers)

Contiguous store non-temporal of bytes from multiple strided vectors (scalar index)

Contiguous store non-temporal of bytes from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

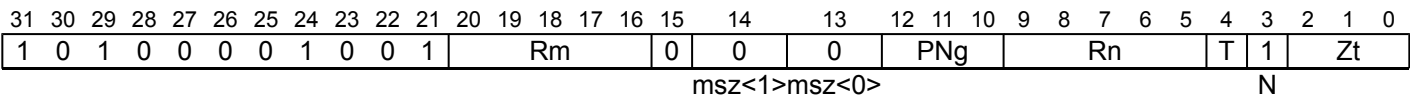
Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

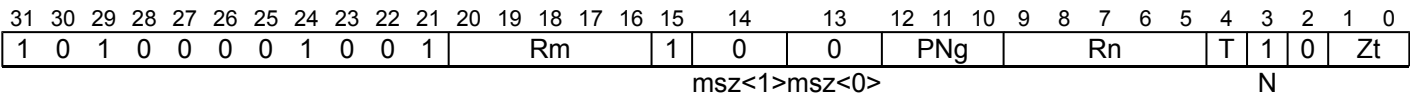


STNT1B { <Zt1>.B, <Zt2>.B }, <PNg>, [<Xn|SP>, <Xm>]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 8;
```

### Four registers

(FEAT\_SME2)



STNT1B { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>, [<Xn|SP>, <Xm>]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 8;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
        t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## STNT1D (scalar plus immediate, strided registers)

Contiguous store non-temporal of doublewords from multiple strided vectors (immediate index)

Contiguous store non-temporal of doublewords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

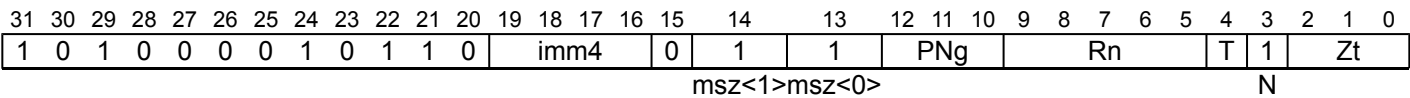
Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

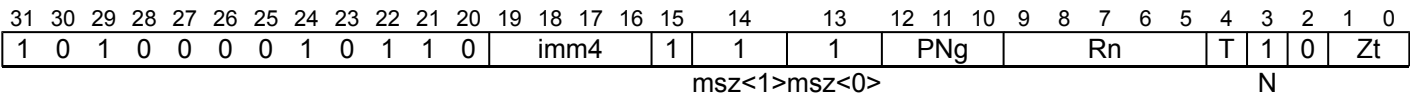


STNT1D { <Zt1>.D, <Zt2>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SME2)



STNT1D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 64;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1>  
For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2>  
For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3>  
Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.  For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
        t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1D (scalar plus scalar, strided registers)

Contiguous store non-temporal of doublewords from multiple strided vectors (scalar index)

Contiguous store non-temporal of doublewords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

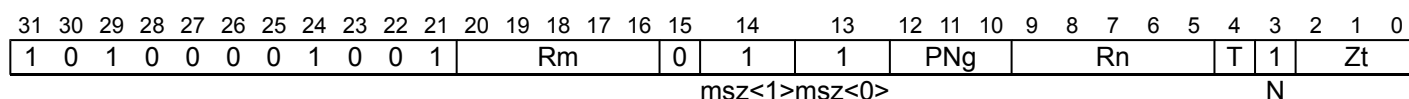
Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

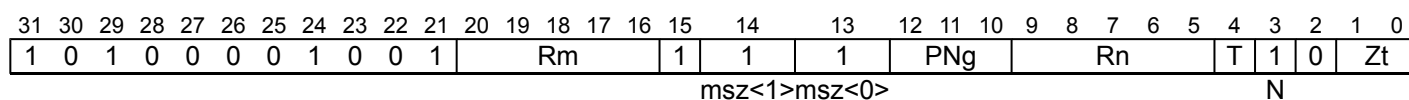


STNT1D { <Zt1>.D, <Zt2>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 64;
```

### Four registers

(FEAT\_SME2)



STNT1D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 64;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
        t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1H (scalar plus immediate, strided registers)

Contiguous store non-temporal of halfwords from multiple strided vectors (immediate index)

Contiguous store non-temporal of halfwords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

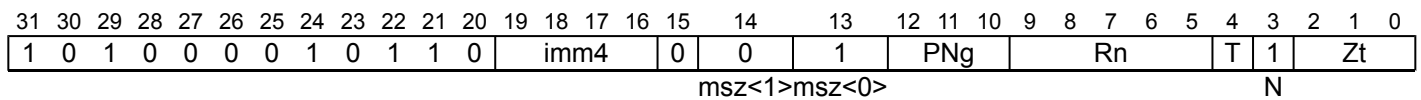
Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

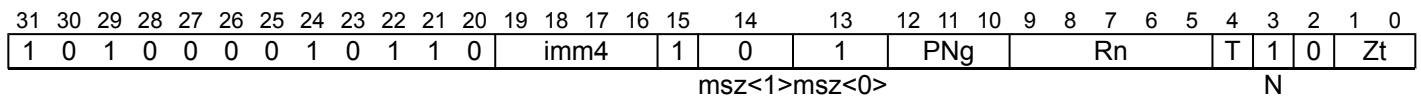


**STNT1H** { <Zt1>.H, <Zt2>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 16;
integer offset = SInt(imm4);
```

### Four registers

(FEAT\_SME2)



**STNT1H** { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 16;
integer offset = SInt(imm4);
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.  For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
        t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1H (scalar plus scalar, strided registers)

Contiguous store non-temporal of halfwords from multiple strided vectors (scalar index)

Contiguous store non-temporal of halfwords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

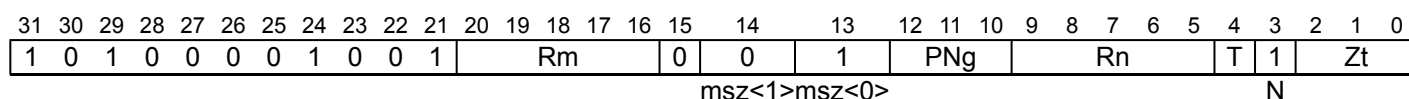
Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

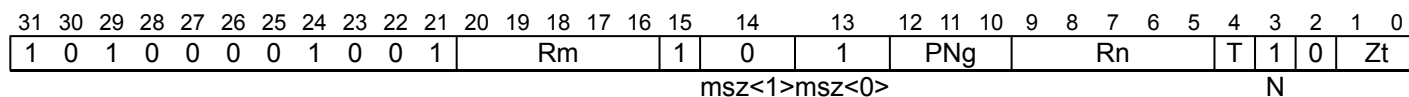


**STNT1H** { <Zt1>.H, <Zt2>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 16;
```

### Four registers

(FEAT\_SME2)



**STNT1H** { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 16;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
        t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



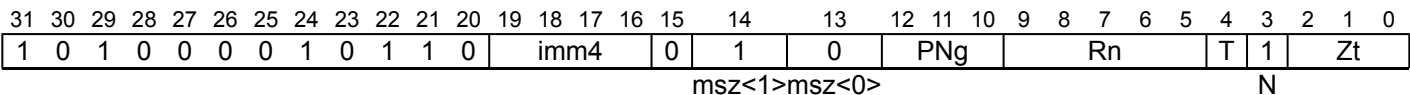
# STNT1W (scalar plus immediate, strided registers)

Contiguous store non-temporal of words from multiple strided vectors (immediate index)

Contiguous store non-temporal of words from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.  
Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.  
It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

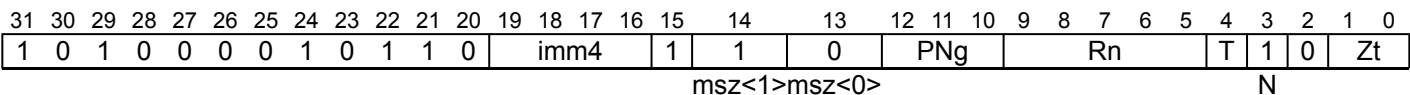
## Two registers (FEAT\_SME2)



STNT1W { <Zt1>.S, <Zt2>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 32;
integer offset = SInt(imm4);
```

## Four registers (FEAT\_SME2)



STNT1W { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 32;
integer offset = SInt(imm4);
```

## Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.  For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
        t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## STNT1W (scalar plus scalar, strided registers)

Contiguous store non-temporal of words from multiple strided vectors (scalar index)

Contiguous store non-temporal of words from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

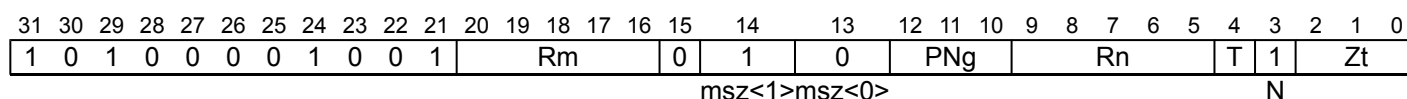
Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

### Two registers

(FEAT\_SME2)

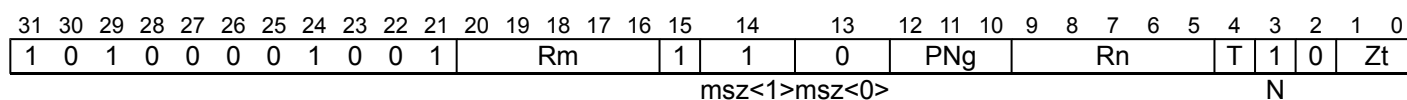


**STNT1W** { <Zt1>.S, <Zt2>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 32;
```

### Four registers

(FEAT\_SME2)



**STNT1W** { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 32;
```

### Assembler Symbols

- <Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".  
For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".
- <Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".  
For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".
- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4>	Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
<PNg>	Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
<Xn SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
        t = t + tstride;

```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STR (vector)

Store ZA array vector

The ZA array vector is selected by the sum of the vector select register and immediate offset, modulo the number of bytes in a Streaming SVE vector. The immediate offset is in the range 0 to 15. The memory address is generated by a 64-bit scalar base, plus the same optional immediate offset multiplied by the current vector length in bytes. This instruction is unpredicated.

The store is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME  
(FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	Rv		0	0	0			Rn			0		off4		

```
STR ZA[<Wv>, <offs>], [<Xn|SP>{, #<offs>, MUL VL}]
```

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer v = UInt('011':Rv);
integer offset = UInt(off4);
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W12-W15, encoded in the "Rv" field.
- <offs> Is the vector select offset and optional memory offset, in the range 0 to 15, defaulting to 0, encoded in the "off4" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

## Operation

```
CheckSMEAndZAEnabled();
constant integer SVL = CurrentSVL;
constant integer dim = SVL DIV 8;
bits(64) base;
integer moffs = offset * dim;
bits(SVL) src;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD dim;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSME(MemOp\_STORE, nontemporal, contiguous, tagchecked);

if HaveTME() && TSTATE.depth > 0 then
    FailTransaction(TMFailure\_ERR, FALSE);

if n == 31 then
    CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

src = ZAvector[vec, SVL];

boolean aligned = IsAligned(base + moffs, 16);

if !aligned && AlignmentEnforced() then
    AArch64.Abort(base + moffs, AlignmentFault(accdesc));

for e = 0 to dim-1
    AArch64.MemSingle[base + moffs, 1, accdesc, aligned] = Elem[src, e, 8];
    moffs = moffs + 1;
```

## Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STR (ZT0)

Store ZT0 register

Store the 64-byte ZT0 register to the memory address provided in the 64-bit scalar base register. This instruction is unpredicated.

The store is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME2  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	1	0	0	1	1	1	1	1	1	0	0	0	0	0	0	Rn				0	0	0	0	0	

STR ZT0, [<Xn|SP>]

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
```

Assembler Symbols

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckSMEEnabled();
CheckSMEZT0Enabled();
constant integer elements = 512 DIV 8;
bits(64) base;
bits(512) table = ZT0[512];
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous, tagchecked);

if HaveTME() && TSTATE.depth > 0 then
    FailTransaction(TMFailure_ERR, FALSE);

if n == 31 then
    CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

boolean aligned = IsAligned(base, 16);

if !aligned && AlignmentEnforced() then
    AArch64.Abort(base, AlignmentFault(accdesc));

for e = 0 to elements-1
    AArch64.MemSingle[base + e, 1, accdesc, aligned] = Elem[table, e, 8];
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

SUB (array accumulators)

Subtract multi-vector from ZA array vector accumulators

Destructively subtract all elements of the two or four source vectors from the corresponding elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

Two ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	0	0	Rv	1	1	1	Zm			0	1	1	off3				
S																															

SUB ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	1	0	Rv	1	1	1	Zm			0	0	1	1	off3			
S																															

SUB ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.



- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.
- For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = ZAvector[vec, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = element1 - element2;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUB (array results, multiple and single vector)

Subtract replicated single vector from multi-vector with ZA array vector results

Subtract all corresponding elements of the second source vector from the two or four first source vectors and place the results in the corresponding elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

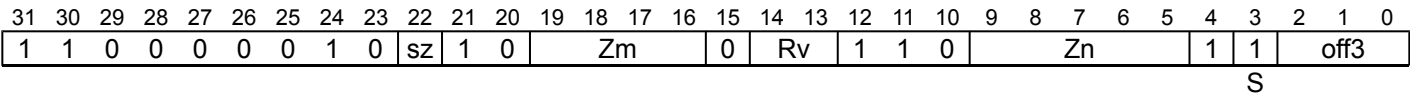
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

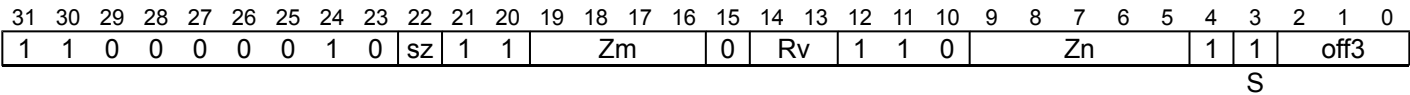
Two ZA single-vectors  
(FEAT\_SME2)



SUB ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors  
(FEAT\_SME2)



SUB ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1>	Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = element1 - element2;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUB (array results, multiple vectors)

Subtract multi-vector from multi-vector with ZA array vector results

Subtract all corresponding elements of the two or four second source vectors from first source vectors and place the results in the corresponding elements of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

Two ZA single-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1		Zm			0	0		Rv	1	1	0		Zn			0	1	1		off3	
																S															

SUB ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm		0		1	0	Rv		1	1	0	Zn		0		0	1	1	off3		
																S															

SUB ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1>	For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm1>	For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
<Zm4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
<Zm2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = element1 - element2;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SUDOT (multiple and indexed vector)

Multi-vector signed by unsigned integer dot-product by indexed elements

The signed by unsigned integer dot product instruction computes the dot product of four signed 8-bit integer values held in each 32-bit element of the two or four first source vectors and four unsigned 8-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

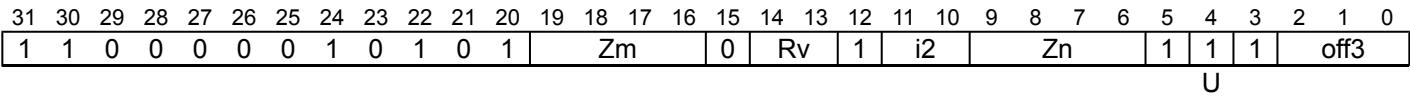
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

### Two ZA single-vectors

(FEAT\_SME2)

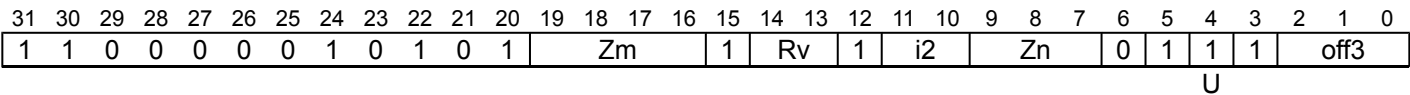


SUDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 2;
```

### Four ZA single-vectors

(FEAT\_SME2)



SUDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 4;
```

### Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> Is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        for i = 0 to 3
            integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# SUDOT (multiple and single vector)

Multi-vector signed by unsigned integer dot-product by vector

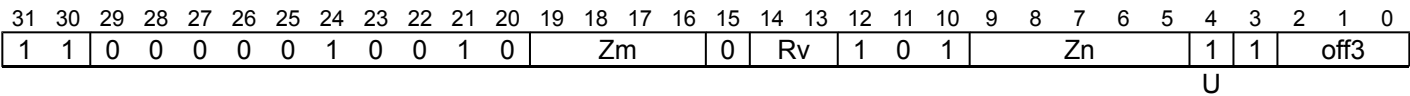
The signed by unsigned integer dot product instruction computes the dot product of four signed 8-bit integer values held in each 32-bit element of the two or four first source vectors and four unsigned 8-bit integer values in the corresponding 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

## Two ZA single-vectors

(FEAT\_SME2)

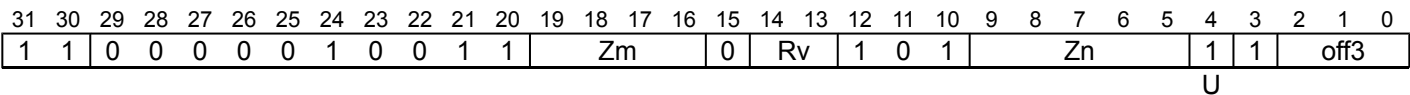


SUDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

## Four ZA single-vectors

(FEAT\_SME2)



SUDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

## Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.



## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUMLALL (multiple and indexed vector)

Multi-vector signed by unsigned integer multiply-add long-long by indexed element

This signed by unsigned integer multiply-add long-long instruction multiplies each signed 8-bit element in the one, two, or four first source vectors with each unsigned 8-bit indexed element of the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The element index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 4 bits. The lowest of the four consecutive vector numbers forming the quad-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: [One ZA quad-vector](#) , [Two ZA quad-vectors](#) and [Four ZA quad-vectors](#)

One ZA quad-vector  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	0	Zm				i4h	Rv		i4l		Zn				1	0	1	off2			
																								U				S			

SUMLALL ZA.S[<Wv>, <offs1>:<offs4>], <Zn>.B, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 1;
```

Two ZA quad-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				0	Rv	0	i4h	Zn				1	1	0	i4l	o1			
																								U		S					

SUMLALL ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 2;
```

Four ZA quad-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				1	Rv		0	i4h		Zn			0	1	1	0	i4l		o1
																								U		S					

```
SUMLALL ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA quad-vector variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.  For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.
<offs4>	For the one ZA quad-vector variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "off2" field times 4 plus 3.  For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	Is the element index, in the range 0 to 15, encoded in the "i4h:i4l" fields.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 3
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer segmentbase = e - (e MOD eltspersegment);
      integer s = 4 * segmentbase + index;
      integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
      integer element2 = UInt(Elem[operand2, s, esize DIV 4]);
      bits(esize) product = (element1 * element2)<esize-1:0>;
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## SUMLALL (multiple and single vector)

Multi-vector signed by unsigned integer multiply-add long-long by vector

This signed by unsigned integer multiply-add long-long instruction multiplies each signed 8-bit element in the two or four first source vectors with each unsigned 8-bit element in the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA quad-vectors](#) and [Four ZA quad-vectors](#)

### Two ZA quad-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			0	Rv		0	0	0	Zn				1	0	1	0	o1		
SZ												U S																			

SUMLALL ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 2;
```

### Four ZA quad-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm			0	Rv		0	0	0	Zn				1	0	1	0	o1		
SZ												U S																			

SUMLALL ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

### Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs1> Is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.
- <offs4> Is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```

CheckStreamingSVEAndZAEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[(n+r) MOD 32, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 3
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
      integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
      bits(esize) product = (element1 * element2)<esize-1:0>;
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUMOPA

Signed by unsigned integer sum of outer products and accumulate

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The signed by unsigned integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds  $SVL_S \times 4$  sub-matrix of signed 8-bit integer values, and the second source holds  $4 \times SVL_S$  sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds  $SVL_D \times 4$  sub-matrix of signed 16-bit integer values, and the second source holds  $4 \times SVL_D$  sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting  $SVL_S \times SVL_S$  widened 32-bit integer or  $SVL_D \times SVL_D$  widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a  $SVL_S \times 4$  sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a  $4 \times SVL_S$  sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a  $SVL_D \times 4$  sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a  $4 \times SVL_D$  sub-matrix.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

32-bit  
(FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	1	0	1	Zm				Pm				Pn				Zn				0	0	0	ZAda	
u0								u1				S																			

SUMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = TRUE;
```

64-bit  
(FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	1	1	1	Zm				Pm				Pn				Zn				0	0	ZAda		
u0								u1				S																			

SUMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = TRUE;
```

## Assembler Symbols

<ZAda>	For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) sum = Elem[operand3, row*dim+col, esize];
    for k = 0 to 3
      if ActivePredicateElement(mask1, 4*row + k, esize DIV 4) &&
         ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
        prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
               Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
        if sub_op then prod = -prod;
        sum = sum + prod;

    Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.





SUMOPS

Signed by unsigned integer sum of outer products and subtract

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The signed by unsigned integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds  $SVL_S \times 4$  sub-matrix of signed 8-bit integer values, and the second source holds  $4 \times SVL_S$  sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds  $SVL_D \times 4$  sub-matrix of signed 16-bit integer values, and the second source holds  $4 \times SVL_D$  sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

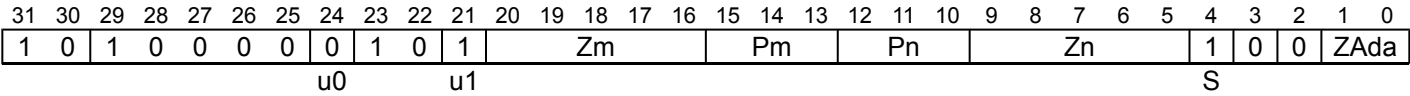
The resulting  $SVL_S \times SVL_S$  widened 32-bit integer or  $SVL_D \times SVL_D$  widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a  $SVL_S \times 4$  sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a  $4 \times SVL_S$  sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a  $SVL_D \times 4$  sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a  $4 \times SVL_D$  sub-matrix.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

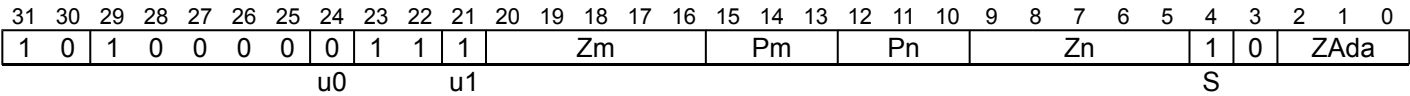
32-bit  
(FEAT\_SME)



SUMOPS [<ZAda>.S](#), [<Pn>/M](#), [<Pm>/M](#), [<Zn>.B](#), [<Zm>.B](#)

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = TRUE;
```

64-bit  
(FEAT\_SME\_I16I64)



SUMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = TRUE;
```

## Assembler Symbols

<ZAda>	For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) sum = Elem[operand3, row*dim+col, esize];
    for k = 0 to 3
      if ActivePredicateElement(mask1, 4*row + k, esize DIV 4) &&
         ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
        prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
               Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
        if sub_op then prod = -prod;
        sum = sum + prod;

    Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.



SUNPK

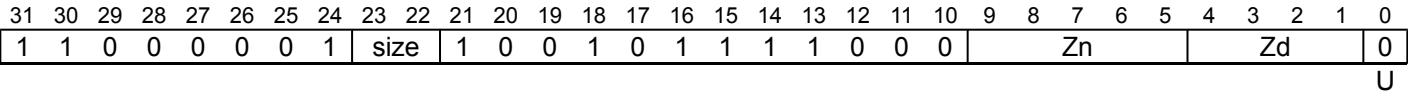
Unpack and sign-extend multi-vector elements

Unpack elements from one or two source vectors and then sign-extend them to place in elements of twice their size within the two or four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

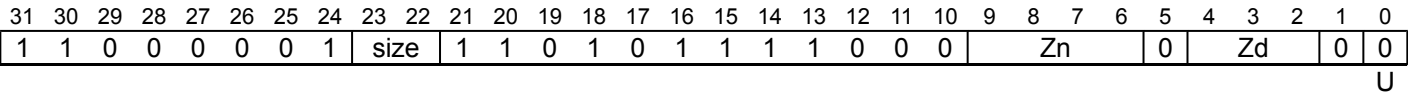
Two registers  
(FEAT\_SME2)



SUNPK { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<Tb>

```
if !HaveSME2() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd:'0');
constant integer nreg = 2;
boolean unsigned = FALSE;
```

Four registers  
(FEAT\_SME2)



SUNPK { <Zd1>.<T>-<Zd4>.<T> }, { <Zn1>.<Tb>-<Zn2>.<Tb> }

```
if !HaveSME2() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn:'0');
integer d = UInt(Zd:'00');
constant integer nreg = 4;
boolean unsigned = FALSE;
```

Assembler Symbols

- <Zd1>
- For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.
- For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
- <Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

### Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer hsize = esize DIV 2;
constant integer sreg = nreg DIV 2;
array [0..3] of bits(VL) results;

for r = 0 to sreg-1
  bits(VL) operand = Z[n+r, VL];
  for i = 0 to 1
    for e = 0 to elements-1
      bits(hsize) element = Elem[operand, i*elements + e, hsize];
      Elem[results[2*r+i], e, esize] = Extend(element, esize, unsigned);

for r = 0 to nreg-1
  Z[d+r, VL] = results[r];

```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUVDOT

Multi-vector signed by unsigned integer vertical dot-product by indexed element

The signed by unsigned integer vertical dot product instruction computes the vertical dot product of the corresponding signed 8-bit elements from the four first source vectors and four unsigned 8-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups.

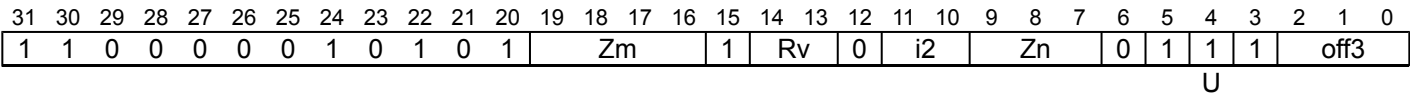
The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits.

The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

The VECTOR GROUP symbol VGx4 indicates that the ZA operand consists of four ZA single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

SME2  
(FEAT\_SME2)



SUVDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> Is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV 4;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for r = 0 to 3
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            bits(VL) operand1 = Z[n+i, VL];
            integer element1 = SInt(Elem[operand1, 4 * e + r, esize DIV 4]);
            integer element2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



SVDOT (2-way)

Multi-vector signed integer vertical dot-product by indexed element

The signed integer vertical dot product instruction computes the vertical dot product of the corresponding two signed 16-bit integer values held in the two first source vectors and two signed 16-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product results are destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits.

The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

The VECTOR GROUP symbol VGx2 indicates that the ZA operand consists of two ZA single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

SME2  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	0	1	Zm			0	Rv		0	i2		Zn			1	0	0	off3			U	

SVDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> Is the immediate index of a group of two 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV 2;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for r = 0 to 1
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 1
            bits(VL) operand1 = Z[n+i, VL];
            integer element1 = SInt(Elem[operand1, 2 * e + r, esize DIV 2]);
            integer element2 = SInt(Elem[operand2, 2 * s + i, esize DIV 2]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# SVDOT (4-way)

Multi-vector signed integer vertical dot-product by indexed element

The signed integer vertical dot product instruction computes the vertical dot product of the corresponding four signed 8-bit or 16-bit integer values held in the four first source vectors and four signed 8-bit or 16-bit integer values in the corresponding indexed 32-bit or 64-bit element of the second source vector. The widened dot product results are destructively added to the corresponding 32-bit or 64-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group.

The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

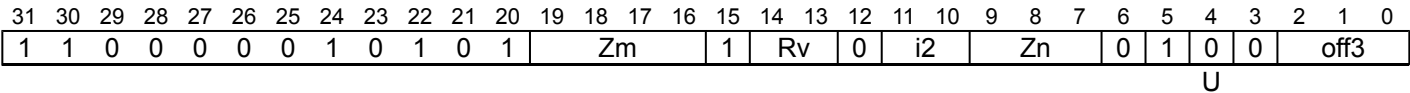
The VECTOR GROUP symbol VGx4 indicates that the ZA operand consists of four ZA single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

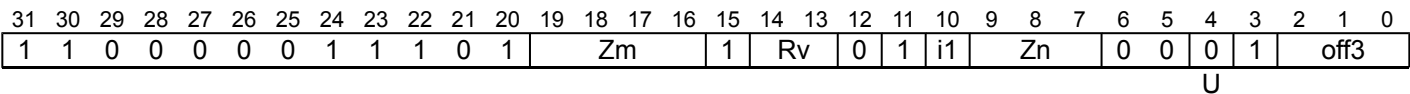
## 32-bit (FEAT\_SME2)



SVDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
```

## 64-bit (FEAT\_SME\_I16I64)



SVDOT ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 64;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i1);
```

## Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> For the 32-bit variant: is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.  
For the 64-bit variant: is the immediate index of a 64-bit group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the "i1" field.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV 4;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for r = 0 to 3
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            bits(VL) operand1 = Z[n+i, VL];
            integer element1 = SInt(Elem[operand1, 4 * e + r, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UCLAMP

Multi-vector unsigned clamp to minimum/maximum vector

Clamp each unsigned element in the two or four destination vectors to between the unsigned minimum value in the corresponding element of the first source vector and the unsigned maximum value in the corresponding element of the second source vector and destructively place the clamped results in the corresponding elements of the two or four destination vectors.

This instruction is unpredicated.  
It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	size		1	Zm					1	1	0	0	0	1	Zn					Zd				1	
																															U	

UCLAMP { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<T>, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd:'0');
constant integer nreg = 2;
```

Four registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size		1	Zm					1	1	0	0	1	1	Zn				Zd		0	1		
																															U

UCLAMP { <Zd1>.<T>-<Zd4>.<T> }, <Zn>.<T>, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd:'00');
constant integer nreg = 4;
```

Assembler Symbols

- <Zd1>

For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.  
  
For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <T>

Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D
- <Zd4>

Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = Z[d+r, VL];
    for e = 0 to elements-1
        integer element1 = UInt(Elem[operand1, e, esize]);
        integer element2 = UInt(Elem[operand2, e, esize]);
        integer element3 = UInt(Elem[operand3, e, esize]);
        integer res = Min(Max(element1, element3), element2);
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

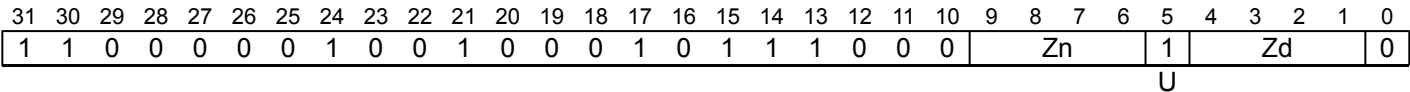
UCVTF

Multi-vector unsigned integer convert to floating-point

Convert to single-precision from unsigned 32-bit integer, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.  
This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.  
It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

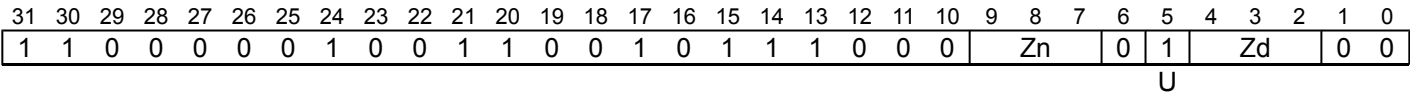
Two registers  
(FEAT\_SME2)



```
UCVTF { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd:'0');
constant integer nreg = 2;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Four registers  
(FEAT\_SME2)



```
UCVTF { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'00');
integer d = UInt(Zd:'00');
constant integer nreg = 4;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

- <Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.  
For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.
- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(32) element = Elem[operand, e, 32];
        Elem[results[r], e, 32] = FixedToFP(element, 0, unsigned, FPCR[], rounding, 32);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## UDOT (2-way, multiple and indexed vector)

Multi-vector unsigned integer dot-product by indexed element

The unsigned integer dot product instruction computes the dot product of two unsigned 16-bit integer values held in each 32-bit element of the two or four first source vectors and two unsigned 16-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

### Two ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	0	1	Zm			0	Rv		1	i2		Zn			0	1	0	off3			U	

UDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 2;
```

### Four ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	0	1	0	1	Zm			1	Rv		1	i2		Zn			0	0	1	0	off3			U	

UDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 4;
```

### Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> Is the immediate index of a group of two 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

## Operation

```
CheckStreamingSVEAndZAEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        for i = 0 to 1
            integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = UInt(Elem[operand2, 2 * s + i, esize DIV 2]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UDOT (2-way, multiple and single vector)

Multi-vector unsigned integer dot-product by vector

The unsigned integer dot product instruction computes the dot product of two unsigned 16-bit integer values held in each 32-bit element of the two or four first source vectors and two unsigned 16-bit integer values in the corresponding 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

### Two ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm			0	Rv		1	0	1	Zn			1	1	off3					
																														U	

UDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

### Four ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	1	Zm			0	Rv		1	0	1	Zn			1	1	off3					
																														U	

UDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

### Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 1
            integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = UInt(Elem[operand2, 2 * e + i, esize DIV 2]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UDOT (2-way, multiple vectors)

Multi-vector unsigned integer dot-product

The unsigned integer dot product instruction computes the dot product of two unsigned 16-bit integer values held in each 32-bit element of the two or four first source vectors and two unsigned 16-bit integer values in the corresponding 32-bit element of the two or four second source vectors. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

### Two ZA single-vectors (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	Zm				0	0	Rv		1	0	1	Zn				0	1	1	off3		
U																															

UDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

### Four ZA single-vectors (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	1	1	Zm				0	1	0	Rv		1	0	1	Zn				0	0	1	1	off3	
U																																

UDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

### Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm1>	For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
<Zm4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
<Zm2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 1
            integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = UInt(Elem[operand2, 2 * e + i, esize DIV 2]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UDOT (4-way, multiple and indexed vector)

Multi-vector unsigned integer dot-product by indexed element

The unsigned integer dot product instruction computes the dot product of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four unsigned 8-bit or 16-bit integer values in the corresponding indexed 32-bit or 64-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unreplicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 4 classes: [Two ZA single-vectors of 32-bit elements](#) , [Two ZA single-vectors of 64-bit elements](#) , [Four ZA single-vectors of 32-bit elements](#) and [Four ZA single-vectors of 64-bit elements](#)

### Two ZA single-vectors of 32-bit elements

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				0	Rv		1	i2	Zn				1	1	0	off3			
U																															

UDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 2;
```

### Two ZA single-vectors of 64-bit elements

(FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				0	Rv		0	0	i1	Zn				0	1	1	off3		
U																															

UDOT ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 64;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i1);
constant integer nreg = 2;
```

## Four ZA single-vectors of 32-bit elements (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				1	Rv		1	i2		Zn		0	1	1	0	off3			
U																															

UDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 4;
```

## Four ZA single-vectors of 64-bit elements (FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				1	Rv		0	0	i1	Zn		0	0	1	1	off3			
U																															

UDOT ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 64;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i1);
constant integer nreg = 4;
```

## Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors of 32-bit elements and two ZA single-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA single-vectors of 32-bit elements and four ZA single-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> For the four ZA single-vectors of 32-bit elements and two ZA single-vectors of 32-bit elements variant: is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.  
For the four ZA single-vectors of 64-bit elements and two ZA single-vectors of 64-bit elements variant: is the immediate index of a 64-bit group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the "i1" field.



## Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        for i = 0 to 3
            integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UDOT (4-way, multiple and single vector)

Multi-vector unsigned integer dot-product by vector

The unsigned integer dot product instruction computes the dot product of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four unsigned 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

Two ZA single-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm			0	Rv		1	0	1	Zn				1	0	off3				
U																															

UDOT ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm			0	Rv		1	0	1	Zn				1	0	off3				
U																															

UDOT ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

- <offs>

Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1>

Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Tb>

Is the size specifier, encoded in "sz":

sz	<Tb>
0	B
1	H
- <Zn4>

Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2>

Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm>

Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
  bits(VL) operand1 = Z[(n+r) MOD 32, VL];
  bits(VL) operand2 = Z[m, VL];
  bits(VL) operand3 = ZAvector[vec, VL];
  for e = 0 to elements-1
    bits(esize) sum = Elem[operand3, e, esize];
    for i = 0 to 3
      integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
      integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
      sum = sum + element1 * element2;
    Elem[result, e, esize] = sum;
  ZAvector[vec, VL] = result;
  vec = vec + vstride;

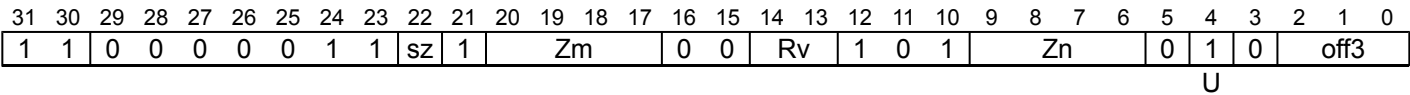
```

UDOT (4-way, multiple vectors)

Multi-vector unsigned integer dot-product

The unsigned integer dot product instruction computes the dot product of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four unsigned 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the two or four second source vectors. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors. The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated. ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented. It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

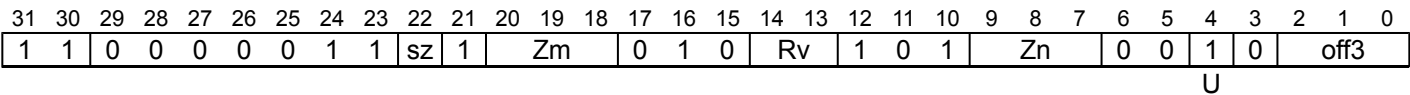
Two ZA single-vectors  
(FEAT\_SME2)



UDOT ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, { <Zm1>.<Tb>-<Zm2>.<Tb> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors  
(FEAT\_SME2)



UDOT ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, { <Zm1>.<Tb>-<Zm4>.<Tb> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Tb> Is the size specifier, encoded in "sz":
- | sz | <Tb> |
|----|------|
| 0  | B    |
| 1  | H    |
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  
For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m+r, VL];
  bits(VL) operand3 = ZAvector[vec, VL];
  for e = 0 to elements-1
    bits(esize) sum = Elem[operand3, e, esize];
    for i = 0 to 3
      integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
      integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
      sum = sum + element1 * element2;
    Elem[result, e, esize] = sum;
  ZAvector[vec, VL] = result;
  vec = vec + vstride;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMAX (multiple and single vector)

Multi-vector unsigned maximum by vector

Determine the unsigned maximum of elements of the second source vector and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	0	0	0	0	0	0	0	0	Zdn				1				
																																U				

UMAX { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt('0':Zm);
constant integer nreg = 2;
boolean unsigned = TRUE;
```

Four registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	1	0	0	0	0	0	0	Zdn				0	1
																																U

UMAX { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt('0':Zm);
constant integer nreg = 4;
boolean unsigned = TRUE;
```

Assembler Symbols

- <Zdn1>
- For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.
- For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        integer element1 = Int(Elem[operand1, e, esize], unsigned);
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer res = Max(element1, element2);
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMAX (multiple vectors)

Multi-vector unsigned maximum

Determine the unsigned maximum of elements of the two or four second source vectors and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	Zm				0	1	0	1	1	0	0	0	0	0	0	0	0	Zdn				1
U																															

UMAX { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt(Zm:'0');
constant integer nreg = 2;
boolean unsigned = TRUE;
```

Four registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	size		1	Zm				0	0	1	0	1	1	1	0	0	0	0	0	Zdn				0	1
U																																

UMAX { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt(Zm:'00');
constant integer nreg = 4;
boolean unsigned = TRUE;
```

Assembler Symbols

- <Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.



<Zdn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
<Zm1>	For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
<Zm4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
<Zm2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        integer element1 = Int(Elem[operand1, e, esize], unsigned);
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer res = Max(element1, element2);
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMIN (multiple and single vector)

Multi-vector unsigned minimum by vector

Determine the unsigned minimum of elements of the second source vector and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0	Zm				1	0	1	0	0	0	0	0	0	0	0	1	Zdn				1

U

UMIN { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt('0':Zm);
constant integer nreg = 2;
boolean unsigned = TRUE;
```

Four registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0	Zm			1	0	1	0	1	0	0	0	0	0	0	1	Zdn			0	1	

U

UMIN { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt('0':Zm);
constant integer nreg = 4;
boolean unsigned = TRUE;
```

Assembler Symbols

- <Zdn1>
- For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.
- For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        integer element1 = Int(Elem[operand1, e, esize], unsigned);
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer res = Min(element1, element2);
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMIN (multiple vectors)

Multi-vector unsigned minimum

Determine the unsigned minimum of elements of the two or four second source vectors and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.  
This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	Zm				0	1	0	1	1	0	0	0	0	0	0	0	1	Zdn				1
U																															

UMIN { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt(Zm:'0');
constant integer nreg = 2;
boolean unsigned = TRUE;
```

Four registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1		Zm	0	0	1	0	1	1	1	0	0	0	0	0	0	0	1		Zdn	0	1	
U																															

UMIN { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt(Zm:'00');
constant integer nreg = 4;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.  
For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
<Zm1>	For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
<Zm4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
<Zm2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        integer element1 = Int(Elem[operand1, e, esize], unsigned);
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer res = Min(element1, element2);
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UMLAL (multiple and indexed vector)

Multi-vector unsigned integer multiply-add long by indexed element

This unsigned integer multiply-add long instruction multiplies each unsigned 16-bit element in the one, two, or four first source vectors with each unsigned 16-bit indexed element of the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the ZA double-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to 7, encoded in 3 bits. The lowest of the two consecutive vector numbers forming the double-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: [One ZA double-vector](#) , [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

### One ZA double-vector

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	0	0	0	0	1	1	1	0	0	Zm				i3h	Rv		1	i3l		Zn				1	0	off3					
																												U		S			

**UMLAL** ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 1;
```

### Two ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	0	0	0	0	1	1	1	0	1	Zm				0	Rv		1	i3h		Zn				0	1	0	i3l	off2			
																												U		S			

**UMLAL** ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 2;
```

### Four ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	0	0	0	0	1	1	1	0	1	Zm				1	Rv		1	i3h		Zn			0	0	1	0	i3l	off2			
																												U		S			

```
UMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
<offs2>	For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 1
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer segmentbase = e - (e MOD eltspersegment);
      integer s = 2 * segmentbase + index;
      integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
      integer element2 = UInt(Elem[operand2, s, esize DIV 2]);
      bits(esize) product = (element1 * element2)<esize-1:0>;
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# UMLAL (multiple and single vector)

Multi-vector unsigned integer multiply-add long by vector

This unsigned integer multiply-add long instruction multiplies each unsigned 16-bit element in the one, two, or four first source vectors with each unsigned 16-bit element in the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated.

It has encodings from 3 classes: [One ZA double-vector](#) , [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

## One ZA double-vector

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	0	0	0	0	1	0	1	1	0	Zm			0	Rv		0	1	1	Zn				1	0	off3						
																												U		S			

UMLAL ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
constant integer nreg = 1;
```

## Two ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	0	0	0	0	1	0	1	1	0	Zm			0	Rv		0	1	0	Zn				1	0	0	off2					
																												U		S			

UMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
constant integer nreg = 2;
```

## Four ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	0	0	0	0	1	0	1	1	1	Zm			0	Rv		0	1	0	Zn				1	0	0	off2					
																												U		S			

```
UMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
<offs2>	For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[(n+r) MOD 32, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 1
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
      integer element2 = UInt(Elem[operand2, 2 * e + i, esize DIV 2]);
      bits(esize) product = (element1 * element2)<esize-1:0>;
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# UMLAL (multiple vectors)

Multi-vector unsigned integer multiply-add long

This unsigned integer multiply-add long instruction multiplies each unsigned 16-bit element in the two or four first source vectors with each unsigned 16-bit element in the two or four second source vectors, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

## Two ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	Zm				0	0	Rv		0	1	0	Zn				0	1	0	0	off2	
																								U				S			

UMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off2:'0');
constant integer nreg = 2;
```

## Four ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	1	1	Zm				0	1	0	Rv		0	1	0	Zn			0	0	1	0	0	off2	
																								U				S				

UMLAL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off2:'0');
constant integer nreg = 4;
```

## Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs1> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
- <offs2> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
- <Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm1>	For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
<Zm4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
<Zm2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = UInt(Elem[operand2, 2 * e + i, esize DIV 2]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UMLALL (multiple and indexed vector)

Multi-vector unsigned integer multiply-add long-long by indexed element

This unsigned integer multiply-add long-long instruction multiplies each unsigned 8-bit or 16-bit element in the one, two, or four first source vectors with each unsigned 8-bit or 16-bit indexed element of second source vector, widens each product to 32-bits or 64-bits and destructively adds these values to the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 3 to 4 bits depending on the size of the element. The lowest of the four consecutive vector numbers forming the quad-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 6 classes: [One ZA quad-vector of 32-bit elements](#) , [One ZA quad-vector of 64-bit elements](#) , [Two ZA quad-vectors of 32-bit elements](#) , [Two ZA quad-vectors of 64-bit elements](#) , [Four ZA quad-vectors of 32-bit elements](#) and [Four ZA quad-vectors of 64-bit elements](#)

### One ZA quad-vector of 32-bit elements

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	0	0	0	0	Zm				i4h	Rv		i4l			Zn				1	0	0	off2			
																														U		S

**UMLALL** ZA.S[<Wv>, <offs1>:<offs4>], <Zn>.B, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 1;
```

### One ZA quad-vector of 64-bit elements

(FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	0	0	0	Zm				i3h	Rv		0	i3l		Zn				1	0	0	off2			
																														U		S

**UMLALL** ZA.D[<Wv>, <offs1>:<offs4>], <Zn>.H, <Zm>.H[<index>]

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 1;
```

## Two ZA quad-vectors of 32-bit elements (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm			0	Rv	0	i4h	Zn			0	1	0	i4l	o1					
																												U		S	

**UMLALL** ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 2;
```

## Two ZA quad-vectors of 64-bit elements (FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm			0	Rv	0	0	i3h	Zn			0	1	0	i3l	o1				
																												U		S	

**UMLALL** ZA.D[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 2;
```

## Four ZA quad-vectors of 32-bit elements (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm			1	Rv		0	i4h		Zn			0	0	1	0	i4l		o1	
																												U		S	

**UMLALL** ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 4;
```

## Four ZA quad-vectors of 64-bit elements (FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm			1	Rv		0	0	i3h	Zn			0	0	1	0	i3l	o1		
																												U S			

```
UMLALL ZA.D[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA quad-vector of 32-bit elements and one ZA quad-vector of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.  For the four ZA quad-vectors of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.
<offs4>	For the one ZA quad-vector of 32-bit elements and one ZA quad-vector of 64-bit elements variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "off2" field times 4 plus 3.  For the four ZA quad-vectors of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	For the two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA quad-vectors of 32-bit elements and four ZA quad-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	For the four ZA quad-vectors of 32-bit elements, one ZA quad-vector of 32-bit elements and two ZA quad-vectors of 32-bit elements variant: is the element index, in the range 0 to 15, encoded in the "i4h:i4l" fields.  For the four ZA quad-vectors of 64-bit elements, one ZA quad-vector of 64-bit elements and two ZA quad-vectors of 64-bit elements variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.



## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 3
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer segmentbase = e - (e MOD eltspersegment);
      integer s = 4 * segmentbase + index;
      integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
      integer element2 = UInt(Elem[operand2, s, esize DIV 4]);
      bits(esize) product = (element1 * element2) < esize-1:0 >;
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# UMLALL (multiple and single vector)

Multi-vector unsigned integer multiply-add long-long by vector

This unsigned integer multiply-add long-long instruction multiplies each unsigned 8-bit or 16-bit element in the one, two, or four first source vectors with each unsigned 8-bit or 16-bit element in the second source vector, widens each product to 32-bits or 64-bits and destructively adds these values to the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 3 classes: [One ZA quad-vector](#) , [Two ZA quad-vectors](#) and [Four ZA quad-vectors](#)

## One ZA quad-vector (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm			0	Rv		0			0	1	Zn				1	0	0	off2	
																											U		S		

UMLALL ZA.<T>[<Wv>, <offs1>:<offs4>], <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
constant integer nreg = 1;
```

## Two ZA quad-vectors (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm			0	Rv		0	0	0	Zn				1	0	0	0	0	1	
																											U	S			

UMLALL ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 2;
```

## Four ZA quad-vectors (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm			0	Rv		0	0	0	Zn				1	0	0	0	0	0	1
																											U	S			

UMLALL ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs1> For the one ZA quad-vector variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.  
For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offs4> For the one ZA quad-vector variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "off2" field times 4 plus 3.  
For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Tb> Is the size specifier, encoded in “sz”:

sz	<Tb>
0	B
1	H

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            bits(esize) product = (element1 * element2) < esize-1:0 >;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMLALL (multiple vectors)

Multi-vector unsigned integer multiply-add long-long

This unsigned integer multiply-add long-long instruction multiplies each unsigned 8-bit or 16-bit element in the two or four first source vectors with each unsigned 8-bit or 16-bit element in the one, two, or four second source vectors, widens each product to 32-bits or 64-bits and destructively adds these values to the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: [Two ZA quad-vectors](#) and [Four ZA quad-vectors](#)

Two ZA quad-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	sz	1			Zm		0	0		Rv	0	0	0			Zn		0	1	0	0	0	o1	
																														U		S

UMLALL ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, { <Zm1>.<Tb>-<Zm2>.<Tb> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(o1:'00');
constant integer nreg = 2;
```

Four ZA quad-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	sz	1		Zm		0	1	0		Rv	0	0	0			Zn		0	0	1	0	0	0	o1
																														U		S

UMLALL ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, { <Zm1>.<Tb>-<Zm4>.<Tb> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>	Is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.						
<offs4>	Is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.						
<Zn1>	For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.						
<Tb>	Is the size specifier, encoded in "sz": <table border="1"> <thead> <tr> <th>sz</th><th>&lt;Tb&gt;</th></tr> </thead> <tbody> <tr> <td>0</td><td>B</td></tr> <tr> <td>1</td><td>H</td></tr> </tbody> </table>	sz	<Tb>	0	B	1	H
sz	<Tb>						
0	B						
1	H						
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.						
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.						
<Zm1>	For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.						
<Zm4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.						
<Zm2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.						

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m+r, VL];
  for i = 0 to 3
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
      integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
      bits(esize) product = (element1 * element2)<esize-1:0>;
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



## UMLSL (multiple and indexed vector)

Multi-vector unsigned integer multiply-subtract long by indexed element

This unsigned integer multiply-subtract long instruction multiplies each unsigned 16-bit element in the one, two, or four first source vectors with each unsigned 16-bit indexed element of the second source vector, widens each product to 32-bits and destructively subtracts these values from the corresponding 32-bit elements of the ZA double-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to 7, encoded in 3 bits. The lowest of the two consecutive vector numbers forming the double-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: [One ZA double-vector](#) , [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

### One ZA double-vector

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	0	Zm				i3h	Rv	1	i3l	Zn				1	1	off3					
																												U		S	

UMLSL ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 1;
```

### Two ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				0	Rv	1	i3h	Zn				0	1	1	i3l	off2			
																												U		S	

UMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 2;
```

### Four ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm				1	Rv	1	i3h	Zn				0	0	1	1	i3l	off2		
																												U		S	



```
UMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
<offs2>	For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 1
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer segmentbase = e - (e MOD eltspersegment);
      integer s = 2 * segmentbase + index;
      integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
      integer element2 = UInt(Elem[operand2, s, esize DIV 2]);
      bits(esize) product = (element1 * element2)<esize-1:0>;
      Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UMLSL (multiple and single vector)

Multi-vector unsigned integer multiply-subtract long by vector

This unsigned integer multiply-subtract long instruction multiplies each unsigned 16-bit element in the one, two, or four first source vectors with each unsigned 16-bit element in the second source vector, widens each product to 32-bits and destructively subtracts these values from the corresponding 32-bit elements of the ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated.

It has encodings from 3 classes: [One ZA double-vector](#) , [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

### One ZA double-vector

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm			0	Rv		0	1	1	Zn			1	1	off3					
																								U		S					

UMLSL ZA.S[<Wv>, <offs1>:<offs2>], <Zn>.H, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
constant integer nreg = 1;
```

### Two ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm			0	Rv		0	1	0	Zn			1	1	0	off2				
																								U		S					

UMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
constant integer nreg = 2;
```

### Four ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	1	Zm			0	Rv		0	1	0	Zn			1	1	0	off2				
																								U		S					

```
UMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.
<offs2>	For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.  For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[(n+r) MOD 32, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 1
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
      integer element2 = UInt(Elem[operand2, 2 * e + i, esize DIV 2]);
      bits(esize) product = (element1 * element2)<esize-1:0>;
      Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UMLS (multiple vectors)

## Multi-vector unsigned integer multiply-subtract long

This unsigned integer multiply-subtract long instruction multiplies each unsigned 16-bit element in the two or four first source vectors with each unsigned 16-bit element in the two or four second source vectors, widens each product to 32-bits and destructively subtracts these values from the corresponding 32-bit elements of the ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The `VECTOR GROUP` symbol, `VGx2` or `VGx4`, indicates that the `ZA` operand consists of two or four `ZA` double-vector groups respectively. The `VECTOR GROUP` symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors and Four ZA double-vectors

## Two ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	Zm			0	0	Rv		0	1	0	Zn			0	1	1	0	off2			
U S																															

```
UMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off2:'0');
constant integer nreg = 2;
```

## Four ZA double-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	0	0	0	0	0	1	1	1	1	Zm		0	1	0	Rv		0	1	0	Zn		0	0	1	1	0	off2						
																										U		S						

```
UMLSL ZA.S[<Wv>, <offs1>:<offs2>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off2:'0');
constant integer nreg = 4;
```

## Assembler Symbols

- |                      |  |
|----------------------|--|
| <b>&lt;Wv&gt;</b>    | Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.  |
| <b>&lt;offs1&gt;</b> | Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.  |
| <b>&lt;offs2&gt;</b> | Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.  |
| <b>&lt;Zn1&gt;</b>   | <p>For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.</p> <p>For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.</p> |

<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm1>	For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
<Zm4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
<Zm2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = UInt(Elem[operand2, 2 * e + i, esize DIV 2]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# UMLSLL (multiple and indexed vector)

Multi-vector unsigned integer multiply-subtract long-long by indexed element

This unsigned integer multiply-subtract long-long instruction multiplies each unsigned 8-bit or 16-bit element in the one, two, or four first source vectors with each unsigned 8-bit or 16-bit indexed element of second source vector, widens each product to 32-bits or 64-bits and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 3 to 4 bits depending on the size of the element. The lowest of the four consecutive vector numbers forming the quad-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 6 classes: [One ZA quad-vector of 32-bit elements](#) , [One ZA quad-vector of 64-bit elements](#) , [Two ZA quad-vectors of 32-bit elements](#) , [Two ZA quad-vectors of 64-bit elements](#) , [Four ZA quad-vectors of 32-bit elements](#) and [Four ZA quad-vectors of 64-bit elements](#)

## One ZA quad-vector of 32-bit elements (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	0	Zm				i4h	Rv		i4l		Zn				1	1	0	off2			
U S																															

UMLSLL ZA.S[<Wv>, <offs1>:<offs4>], <Zn>.B, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 1;
```

## One ZA quad-vector of 64-bit elements (FEAT\_SMEI16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	0	Zm				i3h	Rv		0	i3l		Zn				1	1	0	off2		
U																												S			

UMLSLL ZA.D[<Wv>, <offs1>:<offs4>], <Zn>.H, <Zm>.H[<index>]

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 1;
```



## Two ZA quad-vectors of 32-bit elements (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm			0	Rv	0	i4h	Zn			0	1	1	i4l	o1					
																												U	S		

UMLSLL ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 2;
```

## Two ZA quad-vectors of 64-bit elements (FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm			0	Rv	0	0	i3h	Zn			0	1	1	i3l	o1				
																										U	S				

UMLSLL ZA.D[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 2;
```

## Four ZA quad-vectors of 32-bit elements (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm			1	Rv	0	i4h	Zn			0	0	1	1	i4l	o1				
																												U	S		

UMLSLL ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 4;
```

## Four ZA quad-vectors of 64-bit elements (FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm			1	Rv	0	0	i3h	Zn			0	0	1	1	i3l	o1			
																										U	S				

```
UMLSLL ZA.D[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA quad-vector of 32-bit elements and one ZA quad-vector of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.  For the four ZA quad-vectors of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.
<offs4>	For the one ZA quad-vector of 32-bit elements and one ZA quad-vector of 64-bit elements variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "off2" field times 4 plus 3.  For the four ZA quad-vectors of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	For the two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA quad-vectors of 32-bit elements and four ZA quad-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	For the four ZA quad-vectors of 32-bit elements, one ZA quad-vector of 32-bit elements and two ZA quad-vectors of 32-bit elements variant: is the element index, in the range 0 to 15, encoded in the "i4h:i4l" fields.  For the four ZA quad-vectors of 64-bit elements, one ZA quad-vector of 64-bit elements and two ZA quad-vectors of 64-bit elements variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 3
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer segmentbase = e - (e MOD eltspersegment);
      integer s = 4 * segmentbase + index;
      integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
      integer element2 = UInt(Elem[operand2, s, esize DIV 4]);
      bits(esize) product = (element1 * element2) < esize-1:0>;
      Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UMLSLL (multiple and single vector)

Multi-vector unsigned integer multiply-subtract long-long by vector

This unsigned integer multiply-subtract long-long instruction multiplies each unsigned 8-bit or 16-bit element in the one, two, or four first source vectors with each unsigned 8-bit or 16-bit element in the second source vector, widens each product to 32-bits or 64-bits and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 3 classes: [One ZA quad-vector](#) , [Two ZA quad-vectors](#) and [Four ZA quad-vectors](#)

### One ZA quad-vector

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm		0		Rv		0		0	1	Zn			1		1	0	off2		
																												U	S		

UMLSLL ZA.<T>[<Wv>, <offs1>:<offs4>], <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
constant integer nreg = 1;
```

### Two ZA quad-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm		0		Rv		0			0	0	Zn			1		1	0	0	o1
																												U	S		

UMLSLL ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 2;
```

### Four ZA quad-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm		0		Rv		0			0	0	Zn			1		1	0	0	o1
																												U	S		

UMLSLL ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs1> For the one ZA quad-vector variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.  
  
For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offs4> For the one ZA quad-vector variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "off2" field times 4 plus 3.  
  
For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Tb> Is the size specifier, encoded in “sz”:

sz	<Tb>
0	B
1	H

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            bits(esize) product = (element1 * element2) < esize-1:0 >;
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMLSLL (multiple vectors)

Multi-vector unsigned integer multiply-subtract long-long

This unsigned integer multiply-subtract long-long instruction multiplies each unsigned 8-bit or 16-bit element in the two or four first source vectors with each unsigned 8-bit or 16-bit element in the one, two, or four second source vectors, widens each product to 32-bits or 64-bits and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: [Two ZA quad-vectors](#) and [Four ZA quad-vectors](#)

Two ZA quad-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	sz	1			Zm		0	0		Rv	0	0	0			Zn		0	1	1	0	0	o1	
																														U		S

UMLSLL ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, { <Zm1>.<Tb>-<Zm2>.<Tb> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(o1:'00');
constant integer nreg = 2;
```

Four ZA quad-vectors  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	sz	1		Zm		0	1	0		Rv	0	0	0			Zn		0	0	1	1	0	0	o1
																														U		S

UMLSLL ZA.<T>[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, { <Zm1>.<Tb>-<Zm4>.<Tb> }

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the “Rv” field.

<offs1>	Is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.						
<offs4>	Is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.						
<Zn1>	For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.						
<Tb>	Is the size specifier, encoded in "sz": <table border="1"> <thead> <tr> <th>sz</th><th>&lt;Tb&gt;</th></tr> </thead> <tbody> <tr> <td>0</td><td>B</td></tr> <tr> <td>1</td><td>H</td></tr> </tbody> </table>	sz	<Tb>	0	B	1	H
sz	<Tb>						
0	B						
1	H						
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.						
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.						
<Zm1>	For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.						
<Zm4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.						
<Zm2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.						

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m+r, VL];
  for i = 0 to 3
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
      integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
      bits(esize) product = (element1 * element2)<esize-1:0>;
      Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.





UMOPA (2-way)

Unsigned integer sum of outer products and accumulate

This instruction works with a 32-bit element ZA tile.

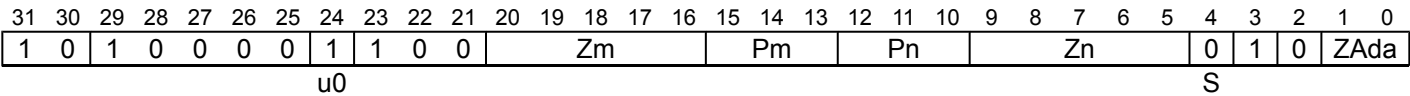
The unsigned integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. The first source holds SVLS×2 sub-matrix of unsigned 16-bit integer values, and the second source holds 2×SVLS sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is inactive, it is treated as having the value 0.

The resulting SVLS×SVLS widened 32-bit integer sum of outer products is then destructively added to the 32-bit integer destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a SVLS×2 sub-matrix, and each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a 2×SVLS sub-matrix.

SME2  
(FEAT\_SME2)



```
UMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean unsigned = TRUE;
```

Assembler Symbols

- <ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) sum = Elem[operand3, row*dim+col, esize];
    for k = 0 to 1
      if ActivePredicateElement(mask1, 2*row + k, esize DIV 2) &&
         ActivePredicateElement(mask2, 2*col + k, esize DIV 2) then
        prod = (Int(Elem[operand1, 2*row + k, esize DIV 2], unsigned) *
               Int(Elem[operand2, 2*col + k, esize DIV 2], unsigned));
        if sub_op then prod = -prod;
        sum = sum + prod;

    Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UMOPA (4-way)

Unsigned integer sum of outer products and accumulate

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The unsigned integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds  $SVL_S \times 4$  sub-matrix of unsigned 8-bit integer values, and the second source holds  $4 \times SVL_S$  sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds  $SVL_D \times 4$  sub-matrix of unsigned 16-bit integer values, and the second source holds  $4 \times SVL_D$  sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is inactive, it is treated as having the value 0.

The resulting  $SVL_S \times SVL_S$  widened 32-bit integer or  $SVL_D \times SVL_D$  widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a  $SVL_S \times 4$  sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a  $4 \times SVL_S$  sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a  $SVL_D \times 4$  sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a  $4 \times SVL_D$  sub-matrix.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

### 32-bit (FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	1	0	1	Zm				Pm				Pn				Zn				0	0	0	ZAda	
u0								u1								S															

UMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;
```

### 64-bit (FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
1	0	1	0	0	0	0	1	1	1	1	Zm				Pm			Pn			Zn				0	0	ZAda										
u0											u1											S															

UMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;
```

## Assembler Symbols

<ZAda>	For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) sum = Elem[operand3, row*dim+col, esize];
    for k = 0 to 3
      if ActivePredicateElement(mask1, 4*row + k, esize DIV 4) &&
         ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
        prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
               Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
        if sub_op then prod = -prod;
        sum = sum + prod;

    Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.



UMOPS (2-way)

Unsigned integer sum of outer products and subtract

This instruction works with a 32-bit element ZA tile.

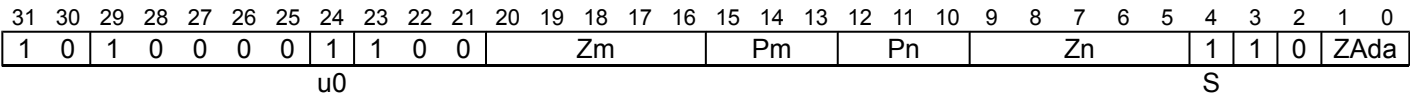
The unsigned integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. The first source holds SVL<sub>S</sub>×2 sub-matrix of unsigned 16-bit integer values, and the second source holds 2×SVL<sub>S</sub> sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is inactive, it is treated as having the value 0.

The resulting SVL<sub>S</sub>×SVL<sub>S</sub> widened 32-bit integer sum of outer products is then destructively subtracted from the 32-bit integer destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a SVL<sub>S</sub>×2 sub-matrix, and each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a 2×SVL<sub>S</sub> sub-matrix.

SME2  
(FEAT\_SME2)



UMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean unsigned = TRUE;
```

Assembler Symbols

- <ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) sum = Elem[operand3, row*dim+col, esize];
    for k = 0 to 1
      if ActivePredicateElement(mask1, 2*row + k, esize DIV 2) &&
         ActivePredicateElement(mask2, 2*col + k, esize DIV 2) then
        prod = (Int(Elem[operand1, 2*row + k, esize DIV 2], unsigned) *
               Int(Elem[operand2, 2*col + k, esize DIV 2], unsigned));
        if sub_op then prod = -prod;
        sum = sum + prod;

    Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



UMOPS (4-way)

Unsigned integer sum of outer products and subtract

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The unsigned integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds SVL<sub>S</sub>×4 sub-matrix of unsigned 8-bit integer values, and the second source holds 4×SVL<sub>S</sub> sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds SVL<sub>D</sub>×4 sub-matrix of unsigned 16-bit integer values, and the second source holds 4×SVL<sub>D</sub> sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

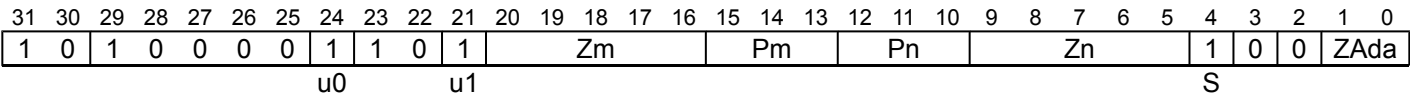
The resulting SVL<sub>S</sub>×SVL<sub>S</sub> widened 32-bit integer or SVL<sub>D</sub>×SVL<sub>D</sub> widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a SVL<sub>S</sub>×4 sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a 4×SVL<sub>S</sub> sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a SVL<sub>D</sub>×4 sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a 4×SVL<sub>D</sub> sub-matrix.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

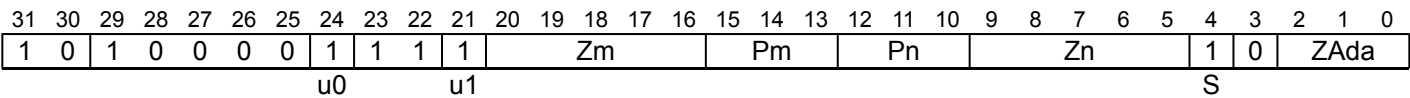
32-bit  
(FEAT\_SME)



UMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;
```

64-bit  
(FEAT\_SME\_I16I64)



UMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;
```

## Assembler Symbols

<ZAda>	For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) sum = Elem[operand3, row*dim+col, esize];
    for k = 0 to 3
      if ActivePredicateElement(mask1, 4*row + k, esize DIV 4) &&
         ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
        prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
               Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
        if sub_op then prod = -prod;
        sum = sum + prod;

    Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

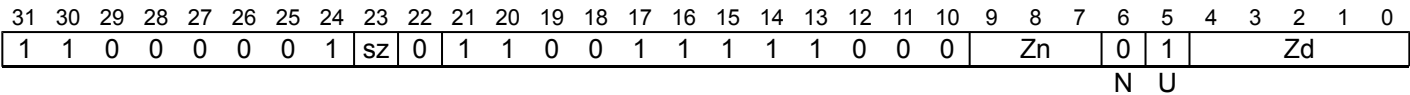


UQCVT (four registers)

Multi-vector unsigned saturating extract narrow

Saturate the unsigned integer value in each element of the four source vectors to quarter the original source element width, and place the results in the quarter-width destination elements.  
This instruction is unpredicated.

SME2  
(FEAT\_SME2)



UQCVT <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(sz);
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <T> Is the size specifier, encoded in "sz":

sz	<T>
0	B
1	H
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	S
1	D
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
bits(VL) result;

for r = 0 to 3
  bits(VL) operand = Z[n+r, VL];
  for e = 0 to elements-1
    integer element = UInt(Elem[operand, e, 4 * esize]);
    Elem[result, r*elements + e, esize] = UnsignedSat(element, esize);

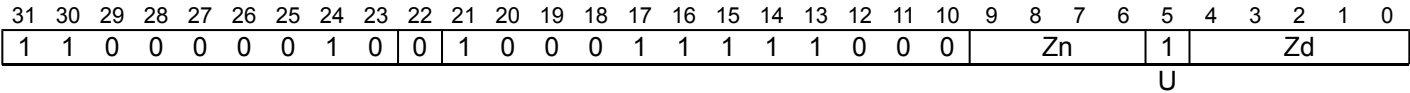
Z[d, VL] = result;
```

UQCVT (two registers)

Multi-vector unsigned saturating extract narrow

Saturate the unsigned integer value in each element of the two source vectors to half the original source element width, and place the results in the half-width destination elements.  
This instruction is unpredicated.

SME2  
(FEAT\_SME2)



```
UQCVT <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;

for r = 0 to 1
  bits(VL) operand = Z[n+r, VL];
  for e = 0 to elements-1
    integer element = UInt(Elem[operand, e, 2 * esize]);
    Elem[result, r*elements + e, esize] = UnsignedSat(element, esize);

Z[d, VL] = result;
```



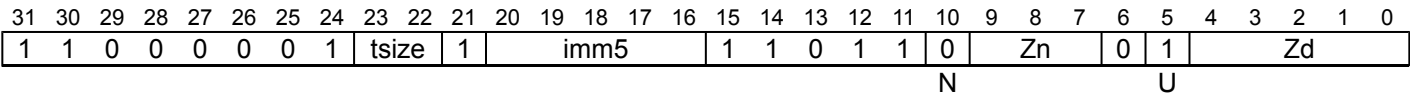
UQRSHR (four registers)

Multi-vector unsigned saturating rounding shift right narrow by immediate

Shift right by an immediate value, the unsigned integer value in each element of the four source vectors and place the rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2  
(FEAT\_SME2)



UQRSHR <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }, #<const>

```
if !HaveSME2() then UNDEFINED;
if tsize == '00' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
integer shift = (8 * esize) - UInt(tsize:imm5);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tsize":

tsize	<T>
00	RESERVED
01	B
1x	H

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "tsize":

tsize	<Tb>
00	RESERVED
01	S
1x	D

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<const> Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in "tsize:imm5".

## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
bits(VL) result;

for r = 0 to 3
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(4 * esize) element = Elem[operand, e, 4 * esize];
        integer res = (UInt(element) + (1 << (shift-1))) >> shift;
        Elem[result, r*elements + e, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

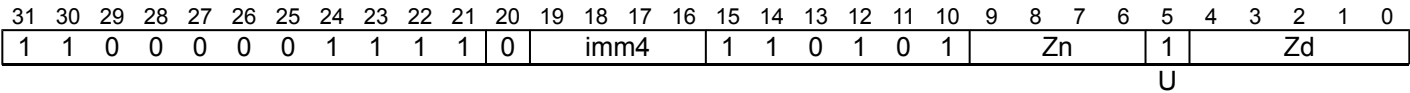


UQRSHR (two registers)

Multi-vector unsigned saturating rounding shift right narrow by immediate

Shift right by an immediate value, the unsigned integer value in each element of the two source vectors and place the rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. The immediate shift amount is an unsigned value in the range 1 to 16. This instruction is unpredicated.

SME2  
(FEAT\_SME2)



```
UQRSHR <Zd>.H, { <Zn1>.S-<Zn2>.S }, #<const>
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
integer shift = esize - UInt(imm4);
```

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <const> Is the immediate shift amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;

for r = 0 to 1
  bits(VL) operand = Z[n+r, VL];
  for e = 0 to elements-1
    bits(2 * esize) element = Elem[operand, e, 2 * esize];
    integer res = (UInt(element) + (1 << (shift-1))) >> shift;
    Elem[result, r*elements + e, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```

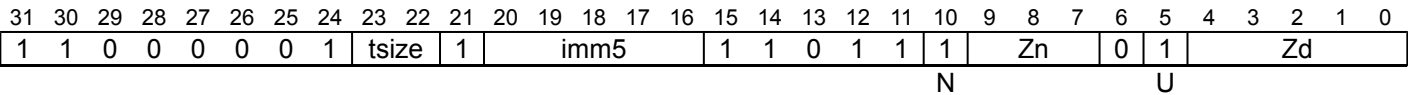
UQRSHRN

Multi-vector unsigned saturating rounding shift right narrow by immediate and interleave

Shift right by an immediate value, the unsigned integer value in each element of the four source vectors and place the four-way interleaved rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's unsigned integer range 0 to (2<sup>N</sup>)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2  
(FEAT\_SME2)



UQRSHRN <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }, #<const>

```
if !HaveSME2() then UNDEFINED;
if tsize == '00' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(tsize);
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
integer shift = (8 * esize) - UInt(tsize:imm5);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tsize":

tsize	<T>
00	RESERVED
01	B
1x	H

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "tsize":

tsize	<Tb>
00	RESERVED
01	S
1x	D

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<const> Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in "tsize:imm5".

## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
bits(VL) result;

for e = 0 to elements-1
    for i = 0 to 3
        bits(VL) operand = Z[n+i, VL];
        bits(4 * esize) element = Elem[operand, e, 4 * esize];
        integer res = (UInt(element) + (1 << (shift-1))) >> shift;
        Elem[result, 4*e + i, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

URSHL (multiple and single vector)

Multi-vector unsigned rounding shift left by vector

Shift the unsigned elements of the two or four first source vectors by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the two or four first source vectors. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	0	0	1	0	0		0	1	Zdn				1
U																																

URSHL { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt('0':Zm);
constant integer nreg = 2;
```

Four registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	1	0	1	0	0	0	1	Zdn				0	1
U																																

URSHL { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt('0':Zm);
constant integer nreg = 4;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        integer element = UInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
        integer res;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = (element + (1 << (shift - 1))) >> shift;
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

URSHL (multiple vectors)

Multi-vector unsigned rounding shift left

Shift the unsigned elements of the two or four first source vectors by corresponding elements of the two or four second source vectors and destructively place the rounded results in the corresponding elements of the two or four first source vectors. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	Zm			0	1	0	1	1	0	0	1	0	0	0	1	Zdn			1			
U																															

URSHL { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'0');
integer m = UInt(Zm:'0');
constant integer nreg = 2;
```

Four registers

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1			Zm		0	0	1	0	1	1	1	0	1	0	0	0	1		Zdn		0	1
U																															

URSHL { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn:'00');
integer m = UInt(Zm:'00');
constant integer nreg = 4;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.
- For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        integer element = UInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
        integer res;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = (element + (1 << (shift - 1))) >> shift;
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# USDOT (multiple and indexed vector)

Multi-vector unsigned by signed integer dot-product by indexed element

The unsigned by signed integer dot product instruction computes the dot product of four unsigned 8-bit integer values held in each 32-bit element of the two or four first source vectors and four signed 8-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

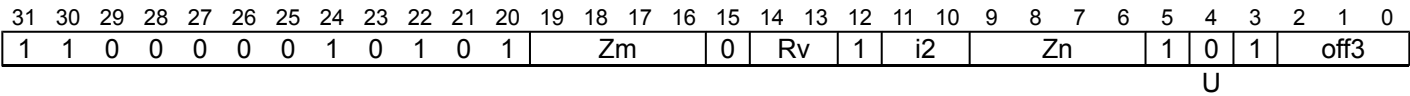
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

## Two ZA single-vectors

(FEAT\_SME2)

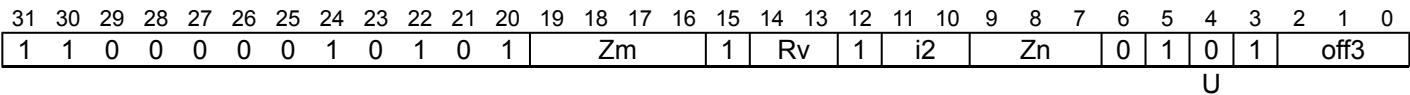


USDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 2;
```

## Four ZA single-vectors

(FEAT\_SME2)



USDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 4;
```

## Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.



For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> Is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        for i = 0 to 3
            integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## USDOT (multiple and single vector)

Multi-vector unsigned by signed integer dot-product by vector

The unsigned by signed integer dot product instruction computes the dot product of four unsigned 8-bit integer values held in each 32-bit element of the two or four first source vectors and four signed 8-bit integer values in the corresponding 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

### Two ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			0	Rv		1	0	1	Zn			0	1	off3					
																														U	

USDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

### Four ZA single-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm			0	Rv		1	0	1	Zn			0	1	off3					
																														U	

USDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

### Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs>	Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
<Zn1>	Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# USDOT (multiple vectors)

Multi-vector unsigned by signed integer dot-product

The unsigned by signed integer dot product instruction computes the dot product of four unsigned 8-bit integer values held in each 32-bit element of the two or four first source vectors and four signed 8-bit integer values in the corresponding 32-bit element of the two or four second source vectors. The widened dot product result is destructively added to corresponding 32-bit element of the ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code. This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

## Two ZA single-vectors (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	0	Rv		1	0	1	Zn			0	0	1	off3				

USDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

## Four ZA single-vectors (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	1	0	Rv			1	0	1	Zn			0	0	0	1	off3	

USDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, { <Zm1>.B-<Zm4>.B }

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

## Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.
- For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## USMLALL (multiple and indexed vector)

Multi-vector unsigned by signed integer multiply-add long-long by indexed element

This unsigned by signed integer multiply-add long-long instruction multiplies each unsigned 8-bit element in the one, two, or four first source vectors with each signed 8-bit indexed element of the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The element index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 4 bits. The lowest of the four consecutive vector numbers forming the quad-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: [One ZA quad-vector](#) , [Two ZA quad-vectors](#) and [Four ZA quad-vectors](#)

### One ZA quad-vector

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	0	Zm				i4h	Rv		i4l		Zn				0	0	1	off2			
																								U				S			

USMLALL ZA.S[<Wv>, <offs1>:<offs4>], <Zn>.B, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 1;
```

### Two ZA quad-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				0	Rv		0	i4h		Zn			1	0	0	i4l		o1	
																												U		S	

USMLALL ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 2;
```

### Four ZA quad-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				1	Rv		0	i4h		Zn			0	1	0	0	i4l		o1
																												U		S	

```
USMLALL ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA quad-vector variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.  For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.
<offs4>	For the one ZA quad-vector variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "off2" field times 4 plus 3.  For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<index>	Is the element index, in the range 0 to 15, encoded in the "i4h:i4l" fields.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[n+r, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 3
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer segmentbase = e - (e MOD eltspersegment);
      integer s = 4 * segmentbase + index;
      integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
      integer element2 = SInt(Elem[operand2, s, esize DIV 4]);
      bits(esize) product = (element1 * element2)<esize-1:0>;
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## USMLALL (multiple and single vector)

Multi-vector unsigned by signed integer multiply-add long-long by vector

This unsigned by signed integer multiply-add long-long instruction multiplies each unsigned 8-bit element in the one, two, or four first source vectors with each signed 8-bit element in the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: [One ZA quad-vector](#) , [Two ZA quad-vectors](#) and [Four ZA quad-vectors](#)

### One ZA quad-vector

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm		0		Rv		0		0	1	Zn			0		0	1	off2		
SZ												U S																			

USMLALL ZA.S[<Wv>, <offs1>:<offs4>], <Zn>.B, <Zm>.B

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
constant integer nreg = 1;
```

### Two ZA quad-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			0	Rv		0	0	0	Zn			0	0	1	0	o1			
SZ												U S																			

USMLALL ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 2;
```

### Four ZA quad-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm		0		Rv		0			0	0	Zn			0		0	1	0	o1
SZ												U S																			

```
USMLALL ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

## Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
<offs1>	For the one ZA quad-vector variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.  For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.
<offs4>	For the one ZA quad-vector variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "off2" field times 4 plus 3.  For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zn1>	Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
<Zm>	Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
  bits(VL) operand1 = Z[(n+r) MOD 32, VL];
  bits(VL) operand2 = Z[m, VL];
  for i = 0 to 3
    bits(VL) operand3 = ZAvector[vec + i, VL];
    for e = 0 to elements-1
      integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
      integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
      bits(esize) product = (element1 * element2)<esize-1:0>;
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    ZAvector[vec + i, VL] = result;
  vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## USMLALL (multiple vectors)

Multi-vector unsigned by signed integer multiply-add long-long

This unsigned by signed integer multiply-add long-long instruction multiplies each unsigned 8-bit element in the two or four first source vectors with each signed 8-bit element in the two or four second source vectors, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: [Two ZA quad-vectors](#) and [Four ZA quad-vectors](#)

### Two ZA quad-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1			Zm		0	0	Rv		0	0	0			Zn		0	0	0	1	0	o1
SZ										U										S											

USMLALL ZA.S[<Wv>, <offs1>:<offs4>{, VGx2}], { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(o1:'00');
constant integer nreg = 2;
```

### Four ZA quad-vectors

(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	0	1	Zm			0	1	0	Rv		0	0	0	Zn			0	0	0	0	0	1	0	o1
SZ										U S																						

USMLALL ZA.S[<Wv>, <offs1>:<offs4>{, VGx4}], { <Zn1>.B-<Zn4>.B }, { <Zm1>.B-<Zm4>.B }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

### Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs1> Is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.
- <offs4> Is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.
- <Zn1> For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.  
For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
<Zm1>	For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.  For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.
<Zm4>	Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
<Zm2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;

```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USMOPA

Unsigned by signed integer sum of outer products and accumulate

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The unsigned by signed integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds  $SVL_S \times 4$  sub-matrix of unsigned 8-bit integer values, and the second source holds  $4 \times SVL_S$  sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds  $SVL_D \times 4$  sub-matrix of unsigned 16-bit integer values, and the second source holds  $4 \times SVL_D$  sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting  $SVL_S \times SVL_S$  widened 32-bit integer or  $SVL_D \times SVL_D$  widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a  $SVL_S \times 4$  sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a  $4 \times SVL_S$  sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a  $SVL_D \times 4$  sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a  $4 \times SVL_D$  sub-matrix.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

32-bit  
(FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	1	0	0	Zm				Pm				Pn				Zn				0	0	0	ZAda	
u0								u1								S															

USMOPA [<ZAda>.S](#), [<Pn>/M](#), [<Pm>/M](#), [<Zn>.B](#), [<Zm>.B](#)

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;
```

64-bit  
(FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	1	1	0	Zm				Pm				Pn				Zn				0	0	ZAda		
u0										u1										S											

USMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;
```

## Assembler Symbols

<ZAda>	For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) sum = Elem[operand3, row*dim+col, esize];
    for k = 0 to 3
      if ActivePredicateElement(mask1, 4*row + k, esize DIV 4) &&
         ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
        prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
               Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
        if sub_op then prod = -prod;
        sum = sum + prod;

    Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.





# USMOPS

Unsigned by signed integer sum of outer products and subtract

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The unsigned by signed integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds  $SVL_S \times 4$  sub-matrix of unsigned 8-bit integer values, and the second source holds  $4 \times SVL_S$  sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds  $SVL_D \times 4$  sub-matrix of unsigned 16-bit integer values, and the second source holds  $4 \times SVL_D$  sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting  $SVL_S \times SVL_S$  widened 32-bit integer or  $SVL_D \times SVL_D$  widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a  $SVL_S \times 4$  sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a  $4 \times SVL_S$  sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a  $SVL_D \times 4$  sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a  $4 \times SVL_D$  sub-matrix.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

## 32-bit (FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	1	0	0	Zm				Pm				Pn				Zn				1	0	0	ZAda	
u0										u1										S											

USMOPS [<ZAda>.S](#), [<Pn>/M](#), [<Pm>/M](#), [<Zn>.B](#), [<Zm>.B](#)

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;
```

## 64-bit (FEAT\_SME\_I16I64)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	1	1	0	Zm				Pm				Pn				Zn				1	0	ZAda		
u0										u1										S											

USMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;
```

## Assembler Symbols

<ZAda>	For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) sum = Elem[operand3, row*dim+col, esize];
    for k = 0 to 3
      if ActivePredicateElement(mask1, 4*row + k, esize DIV 4) &&
         ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
        prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
               Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
        if sub_op then prod = -prod;
        sum = sum + prod;

    Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.



USVDDOT

Multi-vector unsigned by signed integer vertical dot-product by indexed element

The unsigned by signed integer vertical dot product instruction computes the vertical dot product of corresponding unsigned 8-bit elements from the four first source vectors and four signed 8-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the ZA single-vector groups.

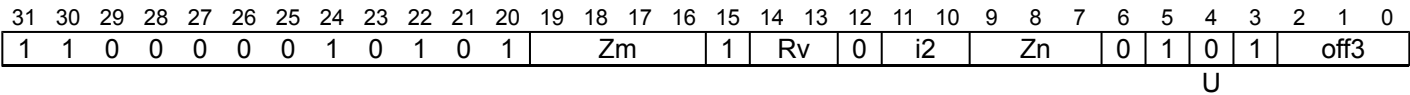
The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits.

The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

The VECTOR GROUP symbol VGx4 indicates that the ZA operand consists of four ZA single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

SME2  
(FEAT\_SME2)



USVDDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> Is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV 4;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for r = 0 to 3
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            bits(VL) operand1 = Z[n+i, VL];
            integer element1 = UInt(Elem[operand1, 4 * e + r, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UUNPK

Unpack and zero-extend multi-vector elements

Unpack elements from one or two source vectors and then zero-extend them to place in elements of twice their size within the two or four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [Two registers](#) and [Four registers](#)

Two registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0	0	1	0	1	1	1	1	0	0	0	Zn				Zd				1		
																															U

UUNPK { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<Tb>

```
if !HaveSME2() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd:'0');
constant integer nreg = 2;
boolean unsigned = TRUE;
```

Four registers  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	1	0	1	0	1	1	1	1	0	0	0	Zn				0	Zd				0	1
																															U

UUNPK { <Zd1>.<T>-<Zd4>.<T> }, { <Zn1>.<Tb>-<Zn2>.<Tb> }

```
if !HaveSME2() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn:'0');
integer d = UInt(Zd:'00');
constant integer nreg = 4;
boolean unsigned = TRUE;
```

Assembler Symbols

- <Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.  
For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
- <Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

### Operation

```

CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer hsize = esize DIV 2;
constant integer sreg = nreg DIV 2;
array [0..3] of bits(VL) results;

for r = 0 to sreg-1
  bits(VL) operand = Z[n+r, VL];
  for i = 0 to 1
    for e = 0 to elements-1
      bits(hsize) element = Elem[operand, i*elements + e, hsize];
      Elem[results[2*r+i], e, esize] = Extend(element, esize, unsigned);

for r = 0 to nreg-1
  Z[d+r, VL] = results[r];

```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UVDOT (2-way)

Multi-vector unsigned integer vertical dot-product by indexed element

The unsigned integer vertical dot product instruction computes the vertical dot product of the corresponding two unsigned 16-bit integer values held in the two first source vectors and two unsigned 16-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product results are destructively added to the corresponding 32-bit element of the ZA single-vector groups.

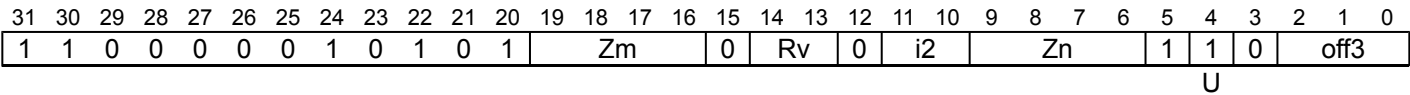
The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits.

The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

The VECTOR GROUP symbol VGx2 indicates that the ZA operand consists of two ZA single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

SME2  
(FEAT\_SME2)



```
UVDOT ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> Is the immediate index of a group of two 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.



## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV 2;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for r = 0 to 1
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 1
            bits(VL) operand1 = Z[n+i, VL];
            integer element1 = UInt(Elem[operand1, 2 * e + r, esize DIV 2]);
            integer element2 = UInt(Elem[operand2, 2 * s + i, esize DIV 2]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## UVDOT (4-way)

Multi-vector unsigned integer vertical dot-product by indexed element

The unsigned integer vertical dot product instruction computes the vertical dot product of the corresponding four unsigned 8-bit or 16-bit integer values held in the four first source vectors and four unsigned 8-bit or 16-bit integer values in the corresponding indexed 32-bit or 64-bit element of the second source vector. The widened dot product results are destructively added to the corresponding 32-bit or 64-bit element of the ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group.

The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

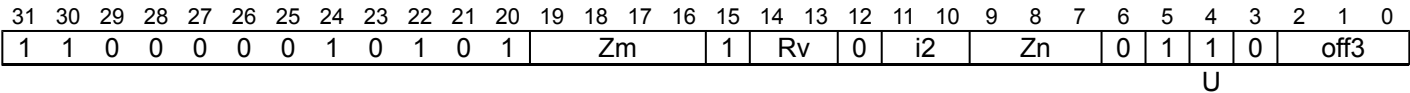
The VECTOR GROUP symbol VGx4 indicates that the ZA operand consists of four ZA single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID\_AA64SMFR0\_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#)

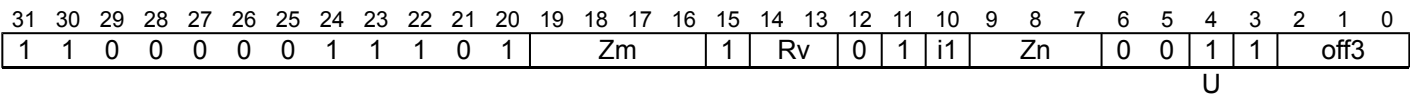
### 32-bit (FEAT\_SME2)



UVDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
```

### 64-bit (FEAT\_SME\_I16I64)



UVDOT ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 64;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i1);
```

### Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> For the 32-bit variant: is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.  
For the 64-bit variant: is the immediate index of a 64-bit group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the "i1" field.

## Operation

```

CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV 4;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for r = 0 to 3
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            bits(VL) operand1 = Z[n+i, VL];
            integer element1 = UInt(Elem[operand1, 4 * e + r, esize DIV 4]);
            integer element2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;

```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UZP (four registers)

Concatenate elements from four vectors

Concatenate every fourth element from each of the four source vectors and place them in the corresponding elements of the four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [8-bit to 64-bit elements](#) and [128-bit element](#)

8-bit to 64-bit elements  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	1	0	1	1	0	1	1	1	0	0	0	Zn	0	0	Zd	1	0					

UZP { <Zd1>.<T>-<Zd4>.<T> }, { <Zn1>.<T>-<Zn4>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn:'00');
integer d = UInt(Zd:'00');
```

128-bit element  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	0	1	1	1	1	1	1	0	0	0	Zn	0	0	Zd	1	0				

UZP { <Zd1>.Q-<Zd4>.Q }, { <Zn1>.Q-<Zn4>.Q }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn:'00');
integer d = UInt(Zd:'00');
```

Assembler Symbols

<Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
if VL < esize * 4 then UNDEFINED;
constant integer quads = VL DIV (esize * 4);
bits(VL) result0;
bits(VL) result1;
bits(VL) result2;
bits(VL) result3;

for r = 0 to 3
    bits(VL) operand = Z[n+r, VL];
    integer base = r * quads;
    for q = 0 to quads-1
        Elem[result0, base+q, esize] = Elem[operand, 4*q+0, esize];
        Elem[result1, base+q, esize] = Elem[operand, 4*q+1, esize];
        Elem[result2, base+q, esize] = Elem[operand, 4*q+2, esize];
        Elem[result3, base+q, esize] = Elem[operand, 4*q+3, esize];

Z[d+0, VL] = result0;
Z[d+1, VL] = result1;
Z[d+2, VL] = result2;
Z[d+3, VL] = result3;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UZP (two registers)

Concatenate elements from two vectors

Concatenate every second element from each of the first and second source vectors and place them in the corresponding elements of the two destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [8-bit to 64-bit elements](#) and [128-bit element](#)

8-bit to 64-bit elements  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1				Zm			1	1	0	1	0	0				Zn				Zd		1

UZP { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<T>, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd:'0');
```

128-bit element  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	Zm					1	1	0	1	0	1	Zn					Zd			1	

UZP { <Zd1>.Q-<Zd2>.Q }, <Zn>.Q, <Zm>.Q

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd:'0');
```

Assembler Symbols

<Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
if VL < esize * 2 then UNDEFINED;
constant integer pairs = VL DIV (esize * 2);
bits(VL) result0;
bits(VL) result1;

for r = 0 to 1
    integer base = r * pairs;
    bits(VL) operand = if r == 0 then Z[n, VL] else Z[m, VL];
    for p = 0 to pairs-1
        Elem[result0, base+p, esize] = Elem[operand, 2*p+0, esize];
        Elem[result1, base+p, esize] = Elem[operand, 2*p+1, esize];

Z[d+0, VL] = result0;
Z[d+1, VL] = result1;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ZERO (double-vector)

### Zero ZA double-vector groups

The instruction zeroes one, two, or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively.

It has encodings from 3 classes: [One ZA double-vector](#) , [Two ZA double-vectors](#) and [Four ZA double-vectors](#)

### One ZA double-vector

(FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	Rv		0	0	0	0	0	0	0	0	0	0	0	off3	

**ZERO** ZA.D[<Wv>, <offs1>:<offs2>]

```
if !HaveSME2p1() then UNDEFINED;
integer v = UInt('010':Rv);
integer offset = UInt(off3:'0');
constant integer ngrp = 1;
constant integer nvec = 2;
```

### Two ZA double-vectors

(FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	Rv		0	0	0	0	0	0	0	0	0	0	0	0	off2

**ZERO** ZA.D[<Wv>, <offs1>:<offs2>, VGx2]

```
if !HaveSME2p1() then UNDEFINED;
integer v = UInt('010':Rv);
integer offset = UInt(off2:'0');
constant integer ngrp = 2;
constant integer nvec = 2;
```

### Four ZA double-vectors

(FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	Rv	0	0	0	0	0	0	0	0	0	0	0	0	off2	

**ZERO** ZA.D[<Wv>, <offs1>:<offs2>, VGx4]

```
if !HaveSME2p1() then UNDEFINED;
integer v = UInt('010':Rv);
integer offset = UInt(off2:'0');
constant integer ngrp = 4;
constant integer nvec = 2;
```

## Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs1> For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.



For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offs2>

For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
integer vectors = VL DIV 8;
integer vstride = vectors DIV ngrp;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
vec = vec - (vec MOD nvec);

for r = 0 to ngrp-1
  for i = 0 to nvec-1
    ZAvector[vec + i, VL] = Zeros(VL);
  vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ZERO (quad-vector)

Zero ZA quad-vector groups

The instruction zeroes one, two, or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within all of, each half of, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively.

It has encodings from 3 classes: [One ZA quad-vector](#) , [Two ZA quad-vectors](#) and [Four ZA quad-vectors](#)

### One ZA quad-vector

(FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	Rv	0	0	0	0	0	0	0	0	0	0	0	0	off2	

**ZERO** ZA.D[<Wv>, <offs1>:<offs4>]

```
if !HaveSME2p1() then UNDEFINED;
integer v = UInt('010':Rv);
integer offset = UInt(off2:'00');
constant integer ngrp = 1;
constant integer nvec = 4;
```

### Two ZA quad-vectors

(FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	Rv	0	0	0	0	0	0	0	0	0	0	0	0	0	01

**ZERO** ZA.D[<Wv>, <offs1>:<offs4>, VGx2]

```
if !HaveSME2p1() then UNDEFINED;
integer v = UInt('010':Rv);
integer offset = UInt(o1:'00');
constant integer ngrp = 2;
constant integer nvec = 4;
```

### Four ZA quad-vectors

(FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	Rv	0	0	0	0	0	0	0	0	0	0	0	0	0	01

**ZERO** ZA.D[<Wv>, <offs1>:<offs4>, VGx4]

```
if !HaveSME2p1() then UNDEFINED;
integer v = UInt('010':Rv);
integer offset = UInt(o1:'00');
constant integer ngrp = 4;
constant integer nvec = 4;
```

## Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs1> For the one ZA quad-vector variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.

For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offs4>

For the one ZA quad-vector variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "off2" field times 4 plus 3.

For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

## Operation

```
CheckStreamingSVEAndZAAEnabled();
constant integer VL = CurrentVL;
integer vectors = VL DIV 8;
integer vstride = vectors DIV ngrp;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
vec = vec - (vec MOD nvec);

for r = 0 to ngrp-1
  for i = 0 to nvec-1
    ZAvector[vec + i, VL] = Zeros(VL);
  vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## ZERO (single-vector)

Zero ZA single-vector groups

The instruction zeroes two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half of or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively.

It has encodings from 2 classes: [Two ZA single-vectors](#) and [Four ZA single-vectors](#)

### Two ZA single-vectors

(FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	Rv		0	0	0	0	0	0	0	0	0	0		off3	

**ZERO** ZA.D[<Wv>, <offs>, VGx2]

```
if !HaveSME2p1() then UNDEFINED;
integer v = UInt('010':Rv);
integer offset = UInt(off3);
constant integer ngrp = 2;
```

### Four ZA single-vectors

(FEAT\_SME2p1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	Rv		0	0	0	0	0	0	0	0	0	0		off3	

**ZERO** ZA.D[<Wv>, <offs>, VGx4]

```
if !HaveSME2p1() then UNDEFINED;
integer v = UInt('010':Rv);
integer offset = UInt(off3);
constant integer ngrp = 4;
```

## Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

## Operation

```
CheckStreamingSVEAndZAEEnabled();
constant integer VL = CurrentVL;
integer vectors = VL DIV 8;
integer vstride = vectors DIV ngrp;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;

for r = 0 to ngrp-1
    ZAvector[vec, VL] = Zeros(VL);
    vec = vec + vstride;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ZERO (tile)

Zero a list of 64-bit element ZA tiles

Zeroes all bytes within each of the up to eight listed 64-bit element tiles named ZA0.D to ZA7.D, leaving the other 64-bit element tiles unmodified.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

For programmer convenience an assembler must also accept the names of 32-bit, 16-bit, and 8-bit element tiles which are converted into the corresponding set of 64-bit element tiles.

In accordance with the architecturally defined mapping between different element size tiles:

- Zeroing the 8-bit element tile name ZA0.B, or the entire array name ZA, is equivalent to zeroing all eight 64-bit element tiles named ZA0.D to ZA7.D.
- Zeroing the 16-bit element tile name ZA0.H is equivalent to zeroing 64-bit element tiles named ZA0.D, ZA2.D, ZA4.D, and ZA6.D.
- Zeroing the 16-bit element tile name ZA1.H is equivalent to zeroing 64-bit element tiles named ZA1.D, ZA3.D, ZA5.D, and ZA7.D.
- Zeroing the 32-bit element tile name ZA0.S is equivalent to zeroing 64-bit element tiles named ZA0.D and ZA4.D.
- Zeroing the 32-bit element tile name ZA1.S is equivalent to zeroing 64-bit element tiles named ZA1.D and ZA5.D.
- Zeroing the 32-bit element tile name ZA2.S is equivalent to zeroing 64-bit element tiles named ZA2.D and ZA6.D.
- Zeroing the 32-bit element tile name ZA3.S is equivalent to zeroing 64-bit element tiles named ZA3.D and ZA7.D.

The preferred disassembly of this instruction uses the shortest list of tile names that represent the encoded immediate mask.

For example:

- An immediate which encodes 64-bit element tiles ZA0.D, ZA1.D, ZA4.D, and ZA5.D is disassembled as {ZA0.S, ZA1.S}.
- An immediate which encodes 64-bit element tiles ZA0.D, ZA2.D, ZA4.D, and ZA6.D is disassembled as {ZA0.H}.
- An all-ones immediate is disassembled as {ZA}.
- An all-zeros immediate is disassembled as an empty list { }.

SME  
(FEAT\_SME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	imm8							

ZERO { <mask> }

```
if !HaveSME() then UNDEFINED;
bits(8) mask = imm8;
constant integer esize = 64;
```

Assembler Symbols

<mask> Is a list of up to eight 64-bit element tile names separated by commas, encoded in the "imm8" field.

## Operation

```
CheckSMEAndZEnabled();  
constant integer SVL = CurrentSVL;  
constant integer dim = SVL DIV esize;  
bits(dim*dim*esize) result = Zeros(dim*dim*esize);  
  
if HaveTME() && TSTATE.depth > 0 then  
    FailTransaction(TMFailure_ERR, FALSE);  
  
for i = 0 to 7  
    if mask<i> == '1' then ZAtile[i, esize, dim*dim*esize] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# ZERO (ZT0)

Zero ZT0

Zero all bytes of the ZT0 register.  
This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

## SME2 (FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

ZERO { ZT0 }

```
if !HaveSME2() then UNDEFINED;
```

### Operation

```
CheckSMEEnabled();
CheckSMEZT0Enabled();

if HaveTME() && TSTATE.depth > 0 then
    FailTransaction(TMFailure_ERR, FALSE);

ZT0[512] = Zeros(512);
```

### Operational information

- If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



ZIP (four registers)

Interleave elements from four vectors

Place the four-way interleaved elements from the four source vectors in the corresponding elements of the four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [8-bit to 64-bit elements](#) and [128-bit element](#)

8-bit to 64-bit elements  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	1	0	1	1	0	1	1	1	0	0	0	Zn	0	0	Zd	0	0					

ZIP { <Zd1>.<T>-<Zd4>.<T> }, { <Zn1>.<T>-<Zn4>.<T> }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn:'00');
integer d = UInt(Zd:'00');
```

128-bit element  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	0	1	1	1	1	1	1	0	0	0	Zn	0	0	Zd	0	0				

ZIP { <Zd1>.Q-<Zd4>.Q }, { <Zn1>.Q-<Zn4>.Q }

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn:'00');
integer d = UInt(Zd:'00');
```

Assembler Symbols

<Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
if VL < esize * 4 then UNDEFINED;
constant integer quads = VL DIV (esize * 4);
bits(VL) operand0 = Z[n, VL];
bits(VL) operand1 = Z[n+1, VL];
bits(VL) operand2 = Z[n+2, VL];
bits(VL) operand3 = Z[n+3, VL];
bits(VL) result;

for r = 0 to 3
    integer base = r * quads;
    for q = 0 to quads-1
        Elem[result, 4*q+0, esize] = Elem[operand0, base+q, esize];
        Elem[result, 4*q+1, esize] = Elem[operand1, base+q, esize];
        Elem[result, 4*q+2, esize] = Elem[operand2, base+q, esize];
        Elem[result, 4*q+3, esize] = Elem[operand3, base+q, esize];
    Z[d+r, VL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ZIP (two registers)

Interleave elements from two vectors

Place the two-way interleaved elements from the first and second source vectors in the corresponding elements of the two destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: [8-bit to 64-bit elements](#) and [128-bit element](#)

8-bit to 64-bit elements  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	Zm						1	1	0	1	0	0	Zn						Zd			0

ZIP { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<T>, <Zm>.<T>

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd:'0');
```

128-bit element  
(FEAT\_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	0	0	1	Zm						1	1	0	1	0	1	Zn						Zd			0

ZIP { <Zd1>.Q-<Zd2>.Q }, <Zn>.Q, <Zm>.Q

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd:'0');
```

Assembler Symbols

<Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

## Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
if VL < esize * 2 then UNDEFINED;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand0 = Z[n, VL];
bits(VL) operand1 = Z[m, VL];
bits(VL) result;

for r = 0 to 1
    integer base = r * pairs;
    for p = 0 to pairs-1
        Elem[result, 2*p+0, esize] = Elem[operand0, base+p, esize];
        Elem[result, 2*p+1, esize] = Elem[operand1, base+p, esize];
    Z[d+r, VL] = result;
```

## Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## Top-level encodings for A64

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0			op1																												

Decode fields		Instruction details
op0	op1	
0	0000	<a href="#">Reserved</a>
1	0000	<a href="#">SME encodings</a>
	0001	UNALLOCATED
	0010	<a href="#">SVE encodings</a>
	0011	UNALLOCATED
	100x	<a href="#">Data Processing -- Immediate</a>
	101x	<a href="#">Branches, Exception Generating and System instructions</a>
	x1x0	<a href="#">Loads and Stores</a>
	x101	<a href="#">Data Processing -- Register</a>
	x111	<a href="#">Data Processing -- Scalar Floating-Point and Advanced SIMD</a>

## Reserved

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0		0000			op1																										

Decode fields		Instruction details
op0	op1	
00	0000000000	<a href="#">UDF</a>
	!= 0000000000	UNALLOCATED
!= 00		UNALLOCATED

## SME encodings

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	op0		0000			op1																op2									

Decode fields			Instruction details
op0	op1	op2	
0x	x10xxxxxxxxxxxxx	xx1x	UNALLOCATED
0x	x11xxxxxxxxxxxxx	x0xx	<a href="#">SME Outer Product - 64 bit</a>
0x	x11xxxxxxxxxxxxx	x1xx	UNALLOCATED
00	x0xxxxxxxxxxxxx		UNALLOCATED
00	x10xxxxxxxxxxxxx	x00x	<a href="#">SME FP Outer Product - 32 bit</a>
00	x10xxxxxxxxxxxxx	x10x	<a href="#">SME2 Outer Product - Misc</a>
01	x10xxxxxxxxxxxxx	xx0x	<a href="#">SME Integer Outer Product - 32 bit</a>
01	00xxxxxxxxxxxxx		<a href="#">SME2 Multi-vector - Memory (Contiguous)</a>
01	10xxxxxxxxxxxxx		<a href="#">SME2 Multi-vector - Memory (Strided)</a>
10	0xx000x0xxxxxxx	0xxx	<a href="#">SME Move into Array</a>
10	0xx000x0xxxxxxx	1xxx	UNALLOCATED
10	0xx000x1xxxxxxx		<a href="#">SME Move from Array</a>

10	0xx010xxxxxxxxxx	x0xx	<a href="#">SME Add Vector to Array</a>
10	0xx010xxxxxxxxxx	x1xx	UNALLOCATED
10	0xx1xxxxxxxxxxxx		UNALLOCATED
10	00x011xxxxxxxxxx		UNALLOCATED
10	0000010xxxxxxxxx		<a href="#">SME Zero</a>
10	0000011xxxxxxxxx		<a href="#">SME2 Multiple Zero</a>
10	0010010xxxxxxxxx		<a href="#">SME2 Zero Lookup Table</a>
10	0010011xxxxxxxxx		<a href="#">SME2 Move Lookup Table</a>
10	01x001xxxxxxxxxx		<a href="#">SME2 Expand Lookup Table (Contiguous)</a>
10	010011xxxxxxxxxx		<a href="#">SME2 Expand Lookup Table (Non-contiguous)</a>
10	011011xxxxxxxxxx		UNALLOCATED
10	1xx00xxxxxxxxxxx		<a href="#">SME2 Multi-vector - Indexed (One register)</a>
10	1xx01xxxx0xxxxxx		<a href="#">SME2 Multi-vector - Indexed (Two registers)</a>
10	1xx01xxxx1xxxxxx		<a href="#">SME2 Multi-vector - Indexed (Four registers)</a>
10	1xx1xxxxx100xxx		<a href="#">SME2 Multi-vector - SVE Select</a>
10	1xx1xxxxx110xxx		<a href="#">SME2 Multi-vector - SVE Constructive Binary</a>
10	1xx1xxxxx111000		<a href="#">SME2 Multi-vector - SVE Constructive Unary</a>
10	1xx1xxxxx111001		UNALLOCATED
10	1xx1xxxxx11101x		UNALLOCATED
10	1xx1xxxxx010110x		<a href="#">SME2 Multi-vector - Multiple Vectors SVE Destructive (Two registers)</a>
10	1xx1xxxxx010111x	xxx0	<a href="#">SME2 Multi-vector - Multiple Vectors SVE Destructive (Four registers)</a>
10	1xx10xxxx10100x		<a href="#">SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers)</a>
10	1xx10xxxx10101x	xxx0	<a href="#">SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers)</a>
10	1xx10xxxx0101x1x	xxx1	UNALLOCATED
10	1xx10xxxx01111xx		UNALLOCATED
10	1xx10xxxx11x11xx		UNALLOCATED
10	1xx10xxxx110101x	xxx1	UNALLOCATED
10	1xx11xxxx1111xx		UNALLOCATED
10	1xx11xxxx01010xx		UNALLOCATED
10	1xx11xxxx010111x	xxx1	UNALLOCATED
10	1xx11xxxx1101xxx		UNALLOCATED
10	10x1xxxxx0xxxxxx		<a href="#">SME2 Multi-vector - Multiple and Single Array Vectors</a>
10	11x1xxxxx00xxxxxx		<a href="#">SME2 Multi-vector - Multiple Array Vectors (Two registers)</a>
10	11x1xxxxx10xxxxxx		<a href="#">SME2 Multi-vector - Multiple Array Vectors (Four registers)</a>
11			<a href="#">SME Memory</a>

## SME Outer Product - 64 bit

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
10	op0			0000			op1	11	op2																			0			

Decode fields			Instruction details
op0	op1	op2	
0	0	0	<a href="#">SME FP64 outer product</a>
0	0	1	UNALLOCATED
0	1		UNALLOCATED
1			<a href="#">SME Int16 outer product</a>

**SME FP64 outer product**

These instructions are under [SME Outer Product - 64 bit](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	1	1	0	Zm					Pm			Pn			Zn					S	0	ZAda		

Decode fields S	Instruction Details	Feature
0	<a href="#">FMOPA (non-widening)</a>	FEAT_SME_F64F64
1	<a href="#">FMOPS (non-widening)</a>	FEAT_SME_F64F64

**SME Int16 outer product**

These instructions are under [SME Outer Product - 64 bit](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	u0	1	1	u1	Zm					Pm			Pn			Zn					S	0	ZAda		

Decode fields u0 u1 S	Instruction Details	Feature
0 0 0	<a href="#">SMOPA (4-way)</a>	FEAT_SME_I16I64
0 0 1	<a href="#">SMOPS (4-way)</a>	FEAT_SME_I16I64
0 1 0	<a href="#">SUMOPA</a>	FEAT_SME_I16I64
0 1 1	<a href="#">SUMOPS</a>	FEAT_SME_I16I64
1 0 0	<a href="#">USMOPA</a>	FEAT_SME_I16I64
1 0 1	<a href="#">USMOPS</a>	FEAT_SME_I16I64
1 1 0	<a href="#">UMOPA (4-way)</a>	FEAT_SME_I16I64
1 1 1	<a href="#">UMOPS (4-way)</a>	FEAT_SME_I16I64

**SME FP Outer Product - 32 bit**

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1000000							op0	10	op1																00						

Decode fields op0 op1	Instruction details
0 0	<a href="#">SME FP32 outer product</a>
0 1	UNALLOCATED
1 0	<a href="#">SME BF16 widening outer product</a>
1 1	<a href="#">SME FP16 widening outer product</a>

**SME FP32 outer product**

These instructions are under [SME FP Outer Product - 32 bit](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	1	0	0	Zm			Pm			Pn			Zn			S	0	0	ZAda					

Decode fields S	Instruction Details	Feature
0	<a href="#">FMOPA (non-widening)</a>	FEAT_SME
1	<a href="#">FMOPS (non-widening)</a>	FEAT_SME

**SME BF16 widening outer product**

These instructions are under [SME FP Outer Product - 32 bit](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	1	0	0	Zm					Pm			Pn			Zn					S	0	0	ZAda	

Decode fields S	Instruction Details	Feature
0	<a href="#">BFMOPA (widening)</a>	FEAT_SME
1	<a href="#">BFMOPS (widening)</a>	FEAT_SME

**SME FP16 widening outer product**

These instructions are under [SME FP Outer Product - 32 bit](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	1	0	1	Zm					Pm			Pn			Zn					S	0	0	ZAda	

Decode fields S	Instruction Details	Feature
0	<a href="#">FMOPA (widening)</a>	FEAT_SME
1	<a href="#">FMOPS (widening)</a>	FEAT_SME

**SME2 Outer Product - Misc**

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
						1000000		op0		10	op1																		10	op2	

Decode fields			Instruction details
op0	op1	op2	
0	0		<a href="#">SME2 32-bit binary outer product</a>
0	1		UNALLOCATED
1	0	0	<a href="#">SME2 FP16 non-widening outer product</a>
1	1	0	<a href="#">SME2 BF16 non-widening outer product</a>
1		1	UNALLOCATED

**SME2 32-bit binary outer product**

These instructions are under [SME2 Outer Product - Misc](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	1	0	0	Zm			Pm			Pn			Zn			S	1	0	ZAda					

Decode fields S	Instruction Details	Feature
0	<a href="#">BMOPA</a>	FEAT_SME2
1	<a href="#">BMOPS</a>	FEAT_SME2

**SME2 FP16 non-widening outer product**

These instructions are under [SME2 Outer Product - Misc](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	1	0	0			Zm			Pm			Pn					Zn		S	1	0	0	ZAda	



Decode fields S	Instruction Details	Feature
0	<a href="#">FMOPA (non-widening)</a>	FEAT_SME_F16F16
1	<a href="#">FMOPS (non-widening)</a>	FEAT_SME_F16F16

**SME2 BF16 non-widening outer product**

These instructions are under [SME2 Outer Product - Misc](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	1	0	1					Zm			Pm			Pn				Zn		S	1	0	0	ZAda

Decode fields S	Instruction Details	Feature
0	<a href="#">BFMOPA (non-widening)</a>	FEAT_SVE_B16B16
1	<a href="#">BFMOPS (non-widening)</a>	FEAT_SVE_B16B16

**SME Integer Outer Product - 32 bit**

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
						1010000			10	op0																		op1	0		

Decode fields op0	Decode fields op1	Instruction details
0	1	<a href="#">SME2 Int16 two-way outer product</a>
1	1	UNALLOCATED
	0	<a href="#">SME Int8 outer product</a>

**SME2 Int16 two-way outer product**

These instructions are under [SME Integer Outer Product - 32 bit](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	u0	1	0	0	Zm			Pm			Pn			Zn				S	1	0	ZAda				

Decode fields u0	Decode fields S	Instruction Details	Feature
0	0	<a href="#">SMOPA (2-way)</a>	FEAT_SME2
0	1	<a href="#">SMOPS (2-way)</a>	FEAT_SME2
1	0	<a href="#">UMOPA (2-way)</a>	FEAT_SME2
1	1	<a href="#">UMOPS (2-way)</a>	FEAT_SME2

**SME Int8 outer product**

These instructions are under [SME Integer Outer Product - 32 bit](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	u0	1	0	u1	Zm			Pm			Pn			Zn			S		0	0	ZAda				

Decode fields u0	Decode fields u1	Decode fields S	Instruction Details	Feature
0	0	0	<a href="#">SMOPA (4-way)</a>	FEAT_SME
0	0	1	<a href="#">SMOPS (4-way)</a>	FEAT_SME

Decode fields			Instruction Details	Feature
u0	u1	S		
0	1	0	<a href="#">SUMOPA</a>	FEAT_SME
0	1	1	<a href="#">SUMOPS</a>	FEAT_SME
1	0	0	<a href="#">USMOPA</a>	FEAT_SME
1	0	1	<a href="#">USMOPS</a>	FEAT_SME
1	1	0	<a href="#">UMOPA (4-way)</a>	FEAT_SME
1	1	1	<a href="#">UMOPS (4-way)</a>	FEAT_SME

## SME2 Multi-vector - Memory (Contiguous)

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
101000000										op0					op1												op2				

Decode fields			Instruction details
op0	op1	op2	
00x	0		<a href="#">SME2 multi-vec contiguous load (scalar plus scalar, two registers)</a>
00x	1	0	<a href="#">SME2 multi-vec contiguous load (scalar plus scalar, four registers)</a>
01x	0		<a href="#">SME2 multi-vec contiguous store (scalar plus scalar, two registers)</a>
01x	1	0	<a href="#">SME2 multi-vec contiguous store (scalar plus scalar, four registers)</a>
0xx	1	1	UNALLOCATED
100	0		<a href="#">SME2 multi-vec contiguous load (scalar plus immediate, two registers)</a>
100	1	0	<a href="#">SME2 multi-vec contiguous load (scalar plus immediate, four registers)</a>
110	0		<a href="#">SME2 multi-vec contiguous store (scalar plus immediate, two registers)</a>
110	1	0	<a href="#">SME2 multi-vec contiguous store (scalar plus immediate, four registers)</a>
1x0	1	1	UNALLOCATED
1x1			UNALLOCATED

## SME2 multi-vec contiguous load (scalar plus scalar, two registers)

These instructions are under [SME2 Multi-vector - Memory \(Contiguous\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm			0	msz	PNg		Rn				Zt			N						

Decode fields		Instruction Details	Feature
msz	N		
00	0	<a href="#">LD1B (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
00	1	<a href="#">LDNT1B (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
01	0	<a href="#">LD1H (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
01	1	<a href="#">LDNT1H (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
10	0	<a href="#">LD1W (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
10	1	<a href="#">LDNT1W (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
11	0	<a href="#">LD1D (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
11	1	<a href="#">LDNT1D (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1

## SME2 multi-vec contiguous load (scalar plus scalar, four registers)

These instructions are under [SME2 Multi-vector - Memory \(Contiguous\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	0	Rm			1	msz		PNg			Rn			Zt			0	N				

Decode fields		Instruction Details	Feature
msz	N		
00	0	<a href="#">LD1B (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
00	1	<a href="#">LDNT1B (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
01	0	<a href="#">LD1H (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
01	1	<a href="#">LDNT1H (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
10	0	<a href="#">LD1W (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
10	1	<a href="#">LDNT1W (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
11	0	<a href="#">LD1D (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
11	1	<a href="#">LDNT1D (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1

**SME2 multi-vec contiguous store (scalar plus scalar, two registers)**

These instructions are under [SME2 Multi-vector - Memory \(Contiguous\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm			0	msz		PNg		Rn			Zt			N						

Decode fields		Instruction Details	Feature
msz	N		
00	0	<a href="#">ST1B (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
00	1	<a href="#">STNT1B (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
01	0	<a href="#">ST1H (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
01	1	<a href="#">STNT1H (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
10	0	<a href="#">ST1W (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
10	1	<a href="#">STNT1W (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
11	0	<a href="#">ST1D (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
11	1	<a href="#">STNT1D (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1

**SME2 multi-vec contiguous store (scalar plus scalar, four registers)**

These instructions are under [SME2 Multi-vector - Memory \(Contiguous\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	1	Rm			1	msz		PNg			Rn			Zt			0	N				

Decode fields		Instruction Details	Feature
msz	N		
00	0	<a href="#">ST1B (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
00	1	<a href="#">STNT1B (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
01	0	<a href="#">ST1H (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
01	1	<a href="#">STNT1H (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
10	0	<a href="#">ST1W (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
10	1	<a href="#">STNT1W (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
11	0	<a href="#">ST1D (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1
11	1	<a href="#">STNT1D (scalar plus scalar, consecutive registers)</a>	FEAT_SVE2p1

**SME2 multi-vec contiguous load (scalar plus immediate, two registers)**

These instructions are under [SME2 Multi-vector - Memory \(Contiguous\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	1	0	0	imm4				0	msz		PNg			Rn				Zt				N	

Decode fields		Instruction Details		Feature
msz	N			
00	0	<a href="#">LD1B (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
00	1	<a href="#">LDNT1B (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
01	0	<a href="#">LD1H (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
01	1	<a href="#">LDNT1H (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
10	0	<a href="#">LD1W (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
10	1	<a href="#">LDNT1W (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
11	0	<a href="#">LD1D (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
11	1	<a href="#">LDNT1D (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1

**SME2 multi-vec contiguous load (scalar plus immediate, four registers)**

These instructions are under [SME2 Multi-vector - Memory \(Contiguous\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	1	0	0	imm4				1	msz		PNg			Rn				Zt			0	N	

Decode fields		Instruction Details		Feature
msz	N			
00	0	<a href="#">LD1B (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
00	1	<a href="#">LDNT1B (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
01	0	<a href="#">LD1H (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
01	1	<a href="#">LDNT1H (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
10	0	<a href="#">LD1W (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
10	1	<a href="#">LDNT1W (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
11	0	<a href="#">LD1D (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
11	1	<a href="#">LDNT1D (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1

**SME2 multi-vec contiguous store (scalar plus immediate, two registers)**

These instructions are under [SME2 Multi-vector - Memory \(Contiguous\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	1	1	0	imm4				0	msz		PNg			Rn				Zt				N	

Decode fields		Instruction Details		Feature
msz	N			
00	0	<a href="#">ST1B (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
00	1	<a href="#">STNT1B (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
01	0	<a href="#">ST1H (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
01	1	<a href="#">STNT1H (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
10	0	<a href="#">ST1W (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
10	1	<a href="#">STNT1W (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
11	0	<a href="#">ST1D (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
11	1	<a href="#">STNT1D (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1

**SME2 multi-vec contiguous store (scalar plus immediate, four registers)**

These instructions are under [SME2 Multi-vector - Memory \(Contiguous\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	1	1	0	imm4			1	msz		PNg			Rn				Zt			0	N		

Decode fields		Instruction Details		Feature
msz	N			
00	0	<a href="#">ST1B (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
00	1	<a href="#">STNT1B (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
01	0	<a href="#">ST1H (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
01	1	<a href="#">STNT1H (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
10	0	<a href="#">ST1W (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
10	1	<a href="#">STNT1W (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
11	0	<a href="#">ST1D (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1
11	1	<a href="#">STNT1D (scalar plus immediate, consecutive registers)</a>		FEAT_SVE2p1

## SME2 Multi-vector - Memory (Strided)

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
101000010									op0						op1												op2				

Decode fields			Instruction details	
op0	op1	op2		
00x	0		<a href="#">SME2 multi-vec non-contiguous load (scalar plus scalar, two registers)</a>	
00x	1	0	<a href="#">SME2 multi-vec non-contiguous load (scalar plus scalar, four registers)</a>	
01x	0		<a href="#">SME2 multi-vec non-contiguous store (scalar plus scalar, two registers)</a>	
01x	1	0	<a href="#">SME2 multi-vec non-contiguous store (scalar plus scalar, four registers)</a>	
0xx	1	1	UNALLOCATED	
100	0		<a href="#">SME2 multi-vec non-contiguous load (scalar plus immediate, two registers)</a>	
100	1	0	<a href="#">SME2 multi-vec non-contiguous load (scalar plus immediate, four registers)</a>	
110	0		<a href="#">SME2 multi-vec non-contiguous store (scalar plus immediate, two registers)</a>	
110	1	0	<a href="#">SME2 multi-vec non-contiguous store (scalar plus immediate, four registers)</a>	
1x0	1	1	UNALLOCATED	
1x1			UNALLOCATED	

## SME2 multi-vec non-contiguous load (scalar plus scalar, two registers)

These instructions are under [SME2 Multi-vector - Memory \(Strided\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	0	Rm				0	msz	PNg			Rn				Zth	N	Ztl					

Decode fields		Instruction Details		Feature
msz	N			
00	0	<a href="#">LD1B (scalar plus scalar, strided registers)</a>		FEAT_SME2
00	1	<a href="#">LDNT1B (scalar plus scalar, strided registers)</a>		FEAT_SME2
01	0	<a href="#">LD1H (scalar plus scalar, strided registers)</a>		FEAT_SME2
01	1	<a href="#">LDNT1H (scalar plus scalar, strided registers)</a>		FEAT_SME2
10	0	<a href="#">LD1W (scalar plus scalar, strided registers)</a>		FEAT_SME2
10	1	<a href="#">LDNT1W (scalar plus scalar, strided registers)</a>		FEAT_SME2
11	0	<a href="#">LD1D (scalar plus scalar, strided registers)</a>		FEAT_SME2
11	1	<a href="#">LDNT1D (scalar plus scalar, strided registers)</a>		FEAT_SME2

**SME2 multi-vec non-contiguous load (scalar plus scalar, four registers)**

These instructions are under [SME2 Multi-vector - Memory \(Strided\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	0					Rm		1		msz		PNg				Rn		Zth	N	0		Ztl

Decode fields		Instruction Details	Feature
msz	N		
00	0	<a href="#">LD1B (scalar plus scalar, strided registers)</a>	FEAT_SME2
00	1	<a href="#">LDNT1B (scalar plus scalar, strided registers)</a>	FEAT_SME2
01	0	<a href="#">LD1H (scalar plus scalar, strided registers)</a>	FEAT_SME2
01	1	<a href="#">LDNT1H (scalar plus scalar, strided registers)</a>	FEAT_SME2
10	0	<a href="#">LD1W (scalar plus scalar, strided registers)</a>	FEAT_SME2
10	1	<a href="#">LDNT1W (scalar plus scalar, strided registers)</a>	FEAT_SME2
11	0	<a href="#">LD1D (scalar plus scalar, strided registers)</a>	FEAT_SME2
11	1	<a href="#">LDNT1D (scalar plus scalar, strided registers)</a>	FEAT_SME2

**SME2 multi-vec non-contiguous store (scalar plus scalar, two registers)**

These instructions are under [SME2 Multi-vector - Memory \(Strided\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	1	Rm			0	msz		PNg			Rn			Zth	N	Ztl						

Decode fields		Instruction Details	Feature
msz	N		
00	0	<a href="#">ST1B (scalar plus scalar, strided registers)</a>	FEAT_SME2
00	1	<a href="#">STNT1B (scalar plus scalar, strided registers)</a>	FEAT_SME2
01	0	<a href="#">ST1H (scalar plus scalar, strided registers)</a>	FEAT_SME2
01	1	<a href="#">STNT1H (scalar plus scalar, strided registers)</a>	FEAT_SME2
10	0	<a href="#">ST1W (scalar plus scalar, strided registers)</a>	FEAT_SME2
10	1	<a href="#">STNT1W (scalar plus scalar, strided registers)</a>	FEAT_SME2
11	0	<a href="#">ST1D (scalar plus scalar, strided registers)</a>	FEAT_SME2
11	1	<a href="#">STNT1D (scalar plus scalar, strided registers)</a>	FEAT_SME2

**SME2 multi-vec non-contiguous store (scalar plus scalar, four registers)**

These instructions are under [SME2 Multi-vector - Memory \(Strided\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	0	1					Rm		1		msz		PNg				Rn		Zth	N	0		Ztl

Decode fields		Instruction Details	Feature
msz	N		
00	0	<a href="#">ST1B (scalar plus scalar, strided registers)</a>	FEAT_SME2
00	1	<a href="#">STNT1B (scalar plus scalar, strided registers)</a>	FEAT_SME2
01	0	<a href="#">ST1H (scalar plus scalar, strided registers)</a>	FEAT_SME2
01	1	<a href="#">STNT1H (scalar plus scalar, strided registers)</a>	FEAT_SME2
10	0	<a href="#">ST1W (scalar plus scalar, strided registers)</a>	FEAT_SME2
10	1	<a href="#">STNT1W (scalar plus scalar, strided registers)</a>	FEAT_SME2
11	0	<a href="#">ST1D (scalar plus scalar, strided registers)</a>	FEAT_SME2
11	1	<a href="#">STNT1D (scalar plus scalar, strided registers)</a>	FEAT_SME2

**SME2 multi-vec non-contiguous load (scalar plus immediate, two registers)**

These instructions are under [SME2 Multi-vector - Memory \(Strided\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	0	0	imm4				0	msz	PNg				Rn				Zth	N	Ztl			

Decode fields		Instruction Details	Feature
msz	N		
00	0	<a href="#">LD1B (scalar plus immediate, strided registers)</a>	FEAT_SME2
00	1	<a href="#">LDNT1B (scalar plus immediate, strided registers)</a>	FEAT_SME2
01	0	<a href="#">LD1H (scalar plus immediate, strided registers)</a>	FEAT_SME2
01	1	<a href="#">LDNT1H (scalar plus immediate, strided registers)</a>	FEAT_SME2
10	0	<a href="#">LD1W (scalar plus immediate, strided registers)</a>	FEAT_SME2
10	1	<a href="#">LDNT1W (scalar plus immediate, strided registers)</a>	FEAT_SME2
11	0	<a href="#">LD1D (scalar plus immediate, strided registers)</a>	FEAT_SME2
11	1	<a href="#">LDNT1D (scalar plus immediate, strided registers)</a>	FEAT_SME2

**SME2 multi-vec non-contiguous load (scalar plus immediate, four registers)**

These instructions are under [SME2 Multi-vector - Memory \(Strided\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	0	0	imm4				1	msz		PNg			Rn				Zth	N	0	Ztl		

Decode fields		Instruction Details	Feature
msz	N		
00	0	<a href="#">LD1B (scalar plus immediate, strided registers)</a>	FEAT_SME2
00	1	<a href="#">LDNT1B (scalar plus immediate, strided registers)</a>	FEAT_SME2
01	0	<a href="#">LD1H (scalar plus immediate, strided registers)</a>	FEAT_SME2
01	1	<a href="#">LDNT1H (scalar plus immediate, strided registers)</a>	FEAT_SME2
10	0	<a href="#">LD1W (scalar plus immediate, strided registers)</a>	FEAT_SME2
10	1	<a href="#">LDNT1W (scalar plus immediate, strided registers)</a>	FEAT_SME2
11	0	<a href="#">LD1D (scalar plus immediate, strided registers)</a>	FEAT_SME2
11	1	<a href="#">LDNT1D (scalar plus immediate, strided registers)</a>	FEAT_SME2

**SME2 multi-vec non-contiguous store (scalar plus immediate, two registers)**

These instructions are under [SME2 Multi-vector - Memory \(Strided\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	1	0	imm4				0	msz		PNg			Rn				Zth	N	Ztl			

Decode fields		Instruction Details	Feature
msz	N		
00	0	<a href="#">ST1B (scalar plus immediate, strided registers)</a>	FEAT_SME2
00	1	<a href="#">STNT1B (scalar plus immediate, strided registers)</a>	FEAT_SME2
01	0	<a href="#">ST1H (scalar plus immediate, strided registers)</a>	FEAT_SME2
01	1	<a href="#">STNT1H (scalar plus immediate, strided registers)</a>	FEAT_SME2
10	0	<a href="#">ST1W (scalar plus immediate, strided registers)</a>	FEAT_SME2
10	1	<a href="#">STNT1W (scalar plus immediate, strided registers)</a>	FEAT_SME2
11	0	<a href="#">ST1D (scalar plus immediate, strided registers)</a>	FEAT_SME2
11	1	<a href="#">STNT1D (scalar plus immediate, strided registers)</a>	FEAT_SME2

**SME2 multi-vec non-contiguous store (scalar plus immediate, four registers)**

These instructions are under [SME2 Multi-vector - Memory \(Strided\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	0	1	1	0	imm4				1	msz	PNg			Rn				Zth	N	0	Ztl			

Decode fields		Instruction Details		Feature
msz	N			
00	0	<a href="#">ST1B (scalar plus immediate, strided registers)</a>		FEAT_SME2
00	1	<a href="#">STNT1B (scalar plus immediate, strided registers)</a>		FEAT_SME2
01	0	<a href="#">ST1H (scalar plus immediate, strided registers)</a>		FEAT_SME2
01	1	<a href="#">STNT1H (scalar plus immediate, strided registers)</a>		FEAT_SME2
10	0	<a href="#">ST1W (scalar plus immediate, strided registers)</a>		FEAT_SME2
10	1	<a href="#">STNT1W (scalar plus immediate, strided registers)</a>		FEAT_SME2
11	0	<a href="#">ST1D (scalar plus immediate, strided registers)</a>		FEAT_SME2
11	1	<a href="#">STNT1D (scalar plus immediate, strided registers)</a>		FEAT_SME2

**SME Move into Array**

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11000000								op0		000	op1	0	op2				op3							op4	0	op5					

Decode fields						Instruction details		Feature
op0	op1	op2	op3	op4	op5			
00	1	00	010	x0	0	<a href="#">MOVA (vector to array, two registers)</a>		FEAT_SME2
00	1	00	011	00	0	<a href="#">MOVA (vector to array, four registers)</a>		FEAT_SME2
00	1	00	0x0	x0	1	UNALLOCATED		-
00	1	00	0x0	x1		UNALLOCATED		-
00	1	00	0x1	00	1	UNALLOCATED		-
00	1	00	0x1	!= 00		UNALLOCATED		-
00	1	01	000	x0	1	UNALLOCATED		-
00	1	01	000	x1		UNALLOCATED		-
00	1	01	001	00	1	UNALLOCATED		-
00	1	01	001	!= 00		UNALLOCATED		-
00	1	01	01x			UNALLOCATED		-
!= 00	1	0x	000	x0	1	UNALLOCATED		-
!= 00	1	0x	000	x1		UNALLOCATED		-
!= 00	1	0x	001	00	1	UNALLOCATED		-
!= 00	1	0x	001	01		UNALLOCATED		-
!= 00	1	0x	001	1x		UNALLOCATED		-
!= 00	1	0x	01x			UNALLOCATED		-
	0					<a href="#">SME move vector to array</a>		-
	1	0x	000	x0	0	<a href="#">SME2 move vector to tile, two registers</a>		-
	1	0x	001	00	0	<a href="#">SME2 move vector to tile, four registers</a>		-
	1	0x	1xx			UNALLOCATED		-
	1	1x				UNALLOCATED		-

**SME move vector to array**

These instructions are under [SME Move into Array](#).



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	size	0	0	0	0	0	Q	V	Rs		Pg						Zn		0			opc		

Decode fields size	Q	Instruction Details	Feature
0x	1	UNALLOCATED	-
00	0	<a href="#">MOVA (vector to tile, single) — 8-bit</a>	FEAT_SME
01	0	<a href="#">MOVA (vector to tile, single) — 16-bit</a>	FEAT_SME
10	0	<a href="#">MOVA (vector to tile, single) — 32-bit</a>	FEAT_SME
10	1	UNALLOCATED	-
11	0	<a href="#">MOVA (vector to tile, single) — 64-bit</a>	FEAT_SME
11	1	<a href="#">MOVA (vector to tile, single) — 128-bit</a>	FEAT_SME

### SME2 move vector to tile, two registers

These instructions are under [SME Move into Array](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	size	0	0	0	1	0	0	V	Rs			0	0	0			Zn		0	0	0		opc	

Decode fields size	Instruction Details	Feature
00	<a href="#">MOVA (vector to tile, two registers) — 8-bit</a>	FEAT_SME2
01	<a href="#">MOVA (vector to tile, two registers) — 16-bit</a>	FEAT_SME2
10	<a href="#">MOVA (vector to tile, two registers) — 32-bit</a>	FEAT_SME2
11	<a href="#">MOVA (vector to tile, two registers) — 64-bit</a>	FEAT_SME2

### SME2 move vector to tile, four registers

These instructions are under [SME Move into Array](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	size	0	0	0	1	0	0	V	Rs			0	0	1			Zn		0	0	0	0		opc

Decode fields size	opc	Instruction Details	Feature
0x	1xx	UNALLOCATED	-
00	0xx	<a href="#">MOVA (vector to tile, four registers) — 8-bit</a>	FEAT_SME2
01	0xx	<a href="#">MOVA (vector to tile, four registers) — 16-bit</a>	FEAT_SME2
10	0xx	<a href="#">MOVA (vector to tile, four registers) — 32-bit</a>	FEAT_SME2
10	1xx	UNALLOCATED	-
11		<a href="#">MOVA (vector to tile, four registers) — 64-bit</a>	FEAT_SME2

### SME Move from Array

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										11000000		op0		000	op1	1	op2			op3		op4									op5

op0	op1	Decode fields op2	op3	op4	op5	Instruction details	Feature
00	1	00	010	00	x0	<a href="#">MOVA (array to vector, two registers)</a>	FEAT_SME2
00	1	00	010	10	x0	<a href="#">MOVAZ (array to vector, two registers)</a>	FEAT_SME2p1
00	1	00	011	00	00	<a href="#">MOVA (array to vector, four registers)</a>	FEAT_SME2

00	1	00	011	10	00	<a href="#">MOVAZ (array to vector, four registers)</a>	FEAT_SME2p1
00	1	00	0x0	x0	x1	UNALLOCATED	-
00	1	00	0x1	x0	!= 00	UNALLOCATED	-
00	1	00	0xx	x1		UNALLOCATED	-
00	1	01	000	x0	x1	UNALLOCATED	-
00	1	01	001	x0	!= 00	UNALLOCATED	-
00	1	01	00x	x1		UNALLOCATED	-
00	1	01	01x			UNALLOCATED	-
!= 00	1	0x	000	x0	x1	UNALLOCATED	-
!= 00	1	0x	001	x0	01	UNALLOCATED	-
!= 00	1	0x	001	x0	1x	UNALLOCATED	-
!= 00	1	0x	00x	x1		UNALLOCATED	-
!= 00	1	0x	01x			UNALLOCATED	-
	0		000	1x		<a href="#">SME zeroing move array to vector</a>	-
	0		!= 000	1x		UNALLOCATED	-
	0			0x		<a href="#">SME move array to vector</a>	-
	1	0x	000	00	x0	<a href="#">SME2 move tile to vector, two registers</a>	-
	1	0x	000	10	x0	<a href="#">SME2 zeroing move tile to vector, two registers</a>	-
	1	0x	001	00	00	<a href="#">SME2 move tile to vector, four registers</a>	-
	1	0x	001	10	00	<a href="#">SME2 zeroing move tile to vector, four registers</a>	-
	1	0x	1xx			UNALLOCATED	-
	1	1x				UNALLOCATED	-

**SME zeroing move array to vector**

These instructions are under [SME Move from Array](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	size		0	0	0	0	1	Q	V	Rs		0	0	0	1		opc					Zd		

Decode fields size	Q	Instruction Details	Feature
0x	1	UNALLOCATED	-
00	0	<a href="#">MOVAZ (tile to vector, single) — 8-bit</a>	FEAT_SME2p1
01	0	<a href="#">MOVAZ (tile to vector, single) — 16-bit</a>	FEAT_SME2p1
10	0	<a href="#">MOVAZ (tile to vector, single) — 32-bit</a>	FEAT_SME2p1
10	1	UNALLOCATED	-
11	0	<a href="#">MOVAZ (tile to vector, single) — 64-bit</a>	FEAT_SME2p1
11	1	<a href="#">MOVAZ (tile to vector, single) — 128-bit</a>	FEAT_SME2p1

**SME move array to vector**

These instructions are under [SME Move from Array](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	size	0	0	0	0	1	Q	V	Rs		Pg		0		opc							Zd		

Decode fields size	Q	Instruction Details	Feature
0x	1	UNALLOCATED	-
00	0	<a href="#">MOVA (tile to vector, single) — 8-bit</a>	FEAT_SME
01	0	<a href="#">MOVA (tile to vector, single) — 16-bit</a>	FEAT_SME
10	0	<a href="#">MOVA (tile to vector, single) — 32-bit</a>	FEAT_SME
10	1	UNALLOCATED	-
11	0	<a href="#">MOVA (tile to vector, single) — 64-bit</a>	FEAT_SME
11	1	<a href="#">MOVA (tile to vector, single) — 128-bit</a>	FEAT_SME

**SME2 move tile to vector, two registers**

These instructions are under [SME Move from Array](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	size	0	0	0	1	1	0	V	Rs		0	0	0	0	0	0	opc				Zd			0

Decode fields size	Instruction Details	Feature
00	<a href="#">MOVA (tile to vector, two registers) — 8-bit</a>	FEAT_SME2
01	<a href="#">MOVA (tile to vector, two registers) — 16-bit</a>	FEAT_SME2
10	<a href="#">MOVA (tile to vector, two registers) — 32-bit</a>	FEAT_SME2
11	<a href="#">MOVA (tile to vector, two registers) — 64-bit</a>	FEAT_SME2

**SME2 zeroing move tile to vector, two registers**

These instructions are under [SME Move from Array](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	size	0	0	0	1	1	0	V	Rs		0	0	0	1	0		opc				Zd			0

Decode fields size	Instruction Details	Feature
00	<a href="#">MOVAZ (tile to vector, two registers) — 8-bit</a>	FEAT_SME2p1
01	<a href="#">MOVAZ (tile to vector, two registers) — 16-bit</a>	FEAT_SME2p1
10	<a href="#">MOVAZ (tile to vector, two registers) — 32-bit</a>	FEAT_SME2p1
11	<a href="#">MOVAZ (tile to vector, two registers) — 64-bit</a>	FEAT_SME2p1

**SME2 move tile to vector, four registers**

These instructions are under [SME Move from Array](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	size	0	0	0	1	1	0	V	Rs		0	0	1	0	0		opc				Zd		0	0

Decode fields size	opc	Instruction Details	Feature
0x	1xx	UNALLOCATED	-
00	0xx	<a href="#">MOVA (tile to vector, four registers) — 8-bit</a>	FEAT_SME2

Decode fields		Instruction Details	Feature
size	opc		
01	0xx	<a href="#">MOVA (tile to vector, four registers) — 16-bit</a>	FEAT_SME2
10	0xx	<a href="#">MOVA (tile to vector, four registers) — 32-bit</a>	FEAT_SME2
10	1xx	UNALLOCATED	-
11		<a href="#">MOVA (tile to vector, four registers) — 64-bit</a>	FEAT_SME2

### SME2 zeroing move tile to vector, four registers

These instructions are under [SME Move from Array](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	size	0	0	0	1	1	0	V	Rs	0	0	1	1	0	opc			Zd			0	0		

Decode fields		Instruction Details	Feature
size	opc		
0x	1xx	UNALLOCATED	-
00	0xx	<a href="#">MOVAZ (tile to vector, four registers) — 8-bit</a>	FEAT_SME2p1
01	0xx	<a href="#">MOVAZ (tile to vector, four registers) — 16-bit</a>	FEAT_SME2p1
10	0xx	<a href="#">MOVAZ (tile to vector, four registers) — 32-bit</a>	FEAT_SME2p1
10	1xx	UNALLOCATED	-
11		<a href="#">MOVAZ (tile to vector, four registers) — 64-bit</a>	FEAT_SME2p1

### SME Add Vector to Array

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11000000								op0		010			op1												op2	0					

Decode fields			Instruction details
op0	op1	op2	
0			UNALLOCATED
1	00	0	<a href="#">SME add vector to array</a>
1	00	1	UNALLOCATED
1	!= 00		UNALLOCATED

### SME add vector to array

These instructions are under [SME Add Vector to Array](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	op	0	1	0	0	0	V		Pm		Pn								Zn		0	0	opc2

Decode fields			Instruction Details	Feature
op	V	opc2		
0		1xx	UNALLOCATED	-
0	0	0xx	<a href="#">ADDHA — 32-bit</a>	FEAT_SME
0	1	0xx	<a href="#">ADDVA — 32-bit</a>	FEAT_SME
1	0		<a href="#">ADDHA — 64-bit</a>	FEAT_SME_I16I64
1	1		<a href="#">ADDVA — 64-bit</a>	FEAT_SME_I16I64

## SME Zero

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11000000000010														op0																	

Decode fields op0	Instruction details	Feature
0000000000	<a href="#">ZERO (tile)</a>	FEAT_SME
!= 0000000000	UNALLOCATED	-

## SME2 Multiple Zero

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11000000000011														op0																	

Decode fields op0	Instruction details
0000000000	<a href="#">SME multiple vectors zero array</a>
!= 0000000000	UNALLOCATED

## SME multiple vectors zero array

These instructions are under [SME2 Multiple Zero](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	1	1	opc		Rv		0	0	0	0	0	0	0	0	0	0	0	0	opc2	

Decode fields opc	Decode fields opc2	Instruction Details	Feature
x1x	1xx	UNALLOCATED	-
000		<a href="#">ZERO (single-vector)</a> — <a href="#">two ZA single-vectors</a>	FEAT_SME2p1
001		<a href="#">ZERO (double-vector)</a> — <a href="#">one ZA double-vector</a>	FEAT_SME2p1
010	0xx	<a href="#">ZERO (double-vector)</a> — <a href="#">two ZA double-vectors</a>	FEAT_SME2p1
011	0xx	<a href="#">ZERO (double-vector)</a> — <a href="#">four ZA double-vectors</a>	FEAT_SME2p1
100		<a href="#">ZERO (single-vector)</a> — <a href="#">four ZA single-vectors</a>	FEAT_SME2p1
101	0xx	<a href="#">ZERO (quad-vector)</a> — <a href="#">one ZA quad-vector</a>	FEAT_SME2p1
101	1xx	UNALLOCATED	-
11x	01x	UNALLOCATED	-
110	00x	<a href="#">ZERO (quad-vector)</a> — <a href="#">two ZA quad-vectors</a>	FEAT_SME2p1
111	00x	<a href="#">ZERO (quad-vector)</a> — <a href="#">four ZA quad-vectors</a>	FEAT_SME2p1

## SME2 Zero Lookup Table

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
11000000010010														op0																							

Decode fields op0	Instruction details
0000000000000000	<a href="#">SME2 zero lookup table</a>
!= 0000000000000000	UNALLOCATED

**SME2 zero lookup table**

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Decode fields opc	Instruction Details	Feature
0000	UNALLOCATED	-
0001	<a href="#">ZERO (ZT0)</a>	FEAT_SME2
001x	UNALLOCATED	-
01xx	UNALLOCATED	-
1xxx	UNALLOCATED	-

**SME2 zero lookup table**

These instructions are under [SME2 Zero Lookup Table](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Decode fields opc	Instruction Details	Feature
0000	UNALLOCATED	-
0001	<a href="#">ZERO (ZT0)</a>	FEAT_SME2
001x	UNALLOCATED	-
01xx	UNALLOCATED	-
1xxx	UNALLOCATED	-

**SME2 Move Lookup Table**

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11000000010011														op0	op1																

Decode fields op0	Decode fields op1	Instruction details
0	00	<a href="#">SME2 move from lookup table</a>
1	00	<a href="#">SME2 move into lookup table</a>
	!= 00	UNALLOCATED

**SME2 move from lookup table**

These instructions are under [SME2 Move Lookup Table](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	1	1	0	0	0	imm3			opc					Rt						

Decode fields opc	Instruction Details	Feature
000xxxx	UNALLOCATED	-
0010xxx	UNALLOCATED	-
00110xx	UNALLOCATED	-

Decode fields opc	Instruction Details	Feature
001110x	UNALLOCATED	-
0011110	UNALLOCATED	-
0011111	<a href="#">MOVT (ZT0 to scalar)</a>	FEAT_SME2
01xxxxx	UNALLOCATED	-
1xxxxxx	UNALLOCATED	-

### SME2 move into lookup table

These instructions are under [SME2 Move Lookup Table](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	1	0	0	1	1	1	0	0	imm3			opc					Rt						

Decode fields opc	Instruction Details	Feature
000xxxx	UNALLOCATED	-
0010xxx	UNALLOCATED	-
00110xx	UNALLOCATED	-
001110x	UNALLOCATED	-
0011110	UNALLOCATED	-
0011111	<a href="#">MOVT (scalar to ZT0)</a>	FEAT_SME2
01xxxxx	UNALLOCATED	-
1xxxxxx	UNALLOCATED	-

### SME2 Expand Lookup Table (Contiguous)

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
110000001									op0	001					op1											op2					

Decode fields			Instruction details
op0	op1	op2	
0	00		UNALLOCATED
0	10	00	<a href="#">SME2 lookup table expand four contiguous registers</a>
0	10	!= 00	UNALLOCATED
0	x1	x0	<a href="#">SME2 lookup table expand two contiguous registers</a>
0	x1	x1	UNALLOCATED
1			<a href="#">SME2 lookup table expand one register</a>

### SME2 lookup table expand four contiguous registers

These instructions are under [SME2 Expand Lookup Table \(Contiguous\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	1	opc		1	0	size	opc2	Zn			Zd			0			0			

Decode fields opc	opc2	Instruction Details	Feature
00x		UNALLOCATED	-
01x	00	<a href="#">LUTI4 (four registers)</a>	FEAT_SME2
01x	01	UNALLOCATED	-

Decode fields opc	opc2	Instruction Details	Feature
01x	1x	UNALLOCATED	-
1xx	00	<a href="#">LUTI2 (four registers)</a>	FEAT_SME2
1xx	01	UNALLOCATED	-
1xx	1x	UNALLOCATED	-

### SME2 lookup table expand two contiguous registers

These instructions are under [SME2 Expand Lookup Table \(Contiguous\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	1	opc			1	size	opc2	Zn			Zd			0						

Decode fields opc	opc2	Instruction Details	Feature
00xx		UNALLOCATED	-
01xx	00	<a href="#">LUTI4 (two registers)</a>	FEAT_SME2
01xx	01	UNALLOCATED	-
01xx	1x	UNALLOCATED	-
1xxx	00	<a href="#">LUTI2 (two registers)</a>	FEAT_SME2
1xxx	01	UNALLOCATED	-
1xxx	1x	UNALLOCATED	-

### SME2 lookup table expand one register

These instructions are under [SME2 Expand Lookup Table \(Contiguous\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	1	opc			size			opc2	Zn			Zd								

Decode fields opc	opc2	Instruction Details	Feature
00xxx		UNALLOCATED	-
01xxx	00	<a href="#">LUTI4 (single)</a>	FEAT_SME2
01xxx	01	UNALLOCATED	-
01xxx	1x	UNALLOCATED	-
1xxxx	00	<a href="#">LUTI2 (single)</a>	FEAT_SME2
1xxxx	01	UNALLOCATED	-
1xxxx	1x	UNALLOCATED	-

### SME2 Expand Lookup Table (Non-contiguous)

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1100000010011													op0													op1					

Decode fields op0	op1	Instruction details
00		UNALLOCATED
10	00	<a href="#">SME2 lookup table expand four non-contiguous registers</a>
10	01	UNALLOCATED
!= 00	1x	UNALLOCATED



x1	0x	<a href="#">SME2 lookup table expand two non-contiguous registers</a>
----	----	---

### SME2 lookup table expand four non-contiguous registers

These instructions are under [SME2 Expand Lookup Table \(Non-contiguous\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	1	1	opc			1	0	size	opc2			Zn				Zdh	0	0	Zdl		

Decode fields opc	opc2	Instruction Details	Feature
00x		UNALLOCATED	-
01x	00	<a href="#">LUTI4 (four registers)</a>	FEAT_SME2p1
01x	01	UNALLOCATED	-
01x	1x	UNALLOCATED	-
1xx	00	<a href="#">LUTI2 (four registers)</a>	FEAT_SME2p1
1xx	01	UNALLOCATED	-
1xx	1x	UNALLOCATED	-

### SME2 lookup table expand two non-contiguous registers

These instructions are under [SME2 Expand Lookup Table \(Non-contiguous\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	1	1	opc			1	size	opc2		Zn				Zdh	0	Zdl					

Decode fields opc	opc2	Instruction Details	Feature
00xx		UNALLOCATED	-
01xx	00	<a href="#">LUTI4 (two registers)</a>	FEAT_SME2p1
01xx	01	UNALLOCATED	-
01xx	1x	UNALLOCATED	-
1xxx	00	<a href="#">LUTI2 (two registers)</a>	FEAT_SME2p1
1xxx	01	UNALLOCATED	-
1xxx	1x	UNALLOCATED	-

### SME2 Multi-vector - Indexed (One register)

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11000001								op0		00		op1								op2											

Decode fields op0	op1	op2	Instruction details
00			<a href="#">SME2 multi-vec indexed long long MLA one source 32-bit</a>
01			UNALLOCATED
10	0	0	<a href="#">SME2 multi-vec indexed long long MLA one source 64-bit</a>
10	0	1	UNALLOCATED
10	1		<a href="#">SME2 multi-vec indexed long FMA one source</a>
11	0		UNALLOCATED
11	1		<a href="#">SME2 multi-vec indexed long MLA one source</a>

**SME2 multi-vec indexed long long MLA one source 32-bit**

These instructions are under [SME2 Multi-vector - Indexed \(One register\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	0	Zm				i4h	Rv	i4l				Zn				U	S	op	off2		

Decode fields			Instruction Details	Feature
U	S	op		
	1	1	UNALLOCATED	-
0	0	0	<a href="#">SMLALL (multiple and indexed vector)</a>	FEAT_SME2
0	0	1	<a href="#">USMLALL (multiple and indexed vector)</a>	FEAT_SME2
0	1	0	<a href="#">SMLSLL (multiple and indexed vector)</a>	FEAT_SME2
1	0	0	<a href="#">UMLALL (multiple and indexed vector)</a>	FEAT_SME2
1	0	1	<a href="#">SUMLALL (multiple and indexed vector)</a>	FEAT_SME2
1	1	0	<a href="#">UMLSLL (multiple and indexed vector)</a>	FEAT_SME2

**SME2 multi-vec indexed long long MLA one source 64-bit**

These instructions are under [SME2 Multi-vector - Indexed \(One register\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	0	Zm				i3h	Rv	0	i3l	Zn				U	S	0	off2				

Decode fields		Instruction Details	Feature
U	S		
0	0	<a href="#">SMLALL (multiple and indexed vector)</a>	FEAT_SME_I16I64
0	1	<a href="#">SMLSLL (multiple and indexed vector)</a>	FEAT_SME_I16I64
1	0	<a href="#">UMLALL (multiple and indexed vector)</a>	FEAT_SME_I16I64
1	1	<a href="#">UMLSLL (multiple and indexed vector)</a>	FEAT_SME_I16I64

**SME2 multi-vec indexed long FMA one source**

These instructions are under [SME2 Multi-vector - Indexed \(One register\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	0	Zm				i3h	Rv	1	i3l	Zn				op	S	off3					

Decode fields		Instruction Details	Feature
op	S		
0	0	<a href="#">FMLAL (multiple and indexed vector)</a>	FEAT_SME2
0	1	<a href="#">FMLSLL (multiple and indexed vector)</a>	FEAT_SME2
1	0	<a href="#">BFMLAL (multiple and indexed vector)</a>	FEAT_SME2
1	1	<a href="#">BFMLSLL (multiple and indexed vector)</a>	FEAT_SME2

**SME2 multi-vec indexed long MLA one source**

These instructions are under [SME2 Multi-vector - Indexed \(One register\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	0	Zm				i3h	Rv	1	i3l	Zn				U	S	off3					

Decode fields		Instruction Details	Feature
U	S		
0	0	<a href="#">SMLAL (multiple and indexed vector)</a>	FEAT_SME2
0	1	<a href="#">SMLSLL (multiple and indexed vector)</a>	FEAT_SME2

Decode fields		Instruction Details	Feature
U	S		
1	0	<a href="#">UMLAL (multiple and indexed vector)</a>	FEAT_SME2
1	1	<a href="#">UMLSL (multiple and indexed vector)</a>	FEAT_SME2

## SME2 Multi-vector - Indexed (Two registers)

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11000001								op0		01						0					op1						op2				

Decode fields			Instruction details
op0	op1	op2	
00	0x		<a href="#">SME2 multi-vec indexed long long MLA two sources 32-bit</a>
00	1x		<a href="#">SME2 multi-vec ternary indexed two registers 16-bit</a>
01			<a href="#">SME2 multi-vec ternary indexed two registers 32-bit</a>
10	00	0	<a href="#">SME2 multi-vec indexed long long MLA two sources 64-bit</a>
10	1x	0	<a href="#">SME2 multi-vec indexed long FMA two sources</a>
11	00	0	<a href="#">SME2 multi-vec ternary indexed two registers 64-bit</a>
11	1x	0	<a href="#">SME2 multi-vec indexed long MLA two sources</a>
1x	00	1	UNALLOCATED
1x	01		UNALLOCATED
1x	1x	1	UNALLOCATED

## SME2 multi-vec indexed long long MLA two sources 32-bit

These instructions are under [SME2 Multi-vector - Indexed \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				0	Rv	0	i4h	Zn				op	U	S	i4l	o1			

Decode fields			Instruction Details	Feature
op	U	S		
0	0	0	<a href="#">SMLALL (multiple and indexed vector)</a>	FEAT_SME2
0	0	1	<a href="#">SMLSLL (multiple and indexed vector)</a>	FEAT_SME2
0	1	0	<a href="#">UMLALL (multiple and indexed vector)</a>	FEAT_SME2
0	1	1	<a href="#">UMLSLL (multiple and indexed vector)</a>	FEAT_SME2
1		1	UNALLOCATED	-
1	0	0	<a href="#">USMLALL (multiple and indexed vector)</a>	FEAT_SME2
1	1	0	<a href="#">SUMLALL (multiple and indexed vector)</a>	FEAT_SME2

## SME2 multi-vec ternary indexed two registers 16-bit

These instructions are under [SME2 Multi-vector - Indexed \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				0	Rv	1	i3h	Zn				op	S	i3l	off3				

Decode fields		Instruction Details	Feature
op	S		
0	0	<a href="#">FMLA (multiple and indexed vector)</a>	FEAT_SME_F16F16
0	1	<a href="#">FMLS (multiple and indexed vector)</a>	FEAT_SME_F16F16
1	0	<a href="#">BFMLA (multiple and indexed vector)</a>	FEAT_SVE_B16B16

Decode fields op	S	Instruction Details	Feature
1	1	<a href="#">BFMLS (multiple and indexed vector)</a>	FEAT_SVE_B16B16

**SME2 multi-vec ternary indexed two registers 32-bit**

These instructions are under [SME2 Multi-vector - Indexed \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	0	1		Zm		0		Rv		op		i2		Zn			opc2			off3		

Decode fields op	opc2	Instruction Details	Feature
0	000	<a href="#">FMLA (multiple and indexed vector)</a>	FEAT_SME2
0	001	<a href="#">FVDOT</a>	FEAT_SME2
0	010	<a href="#">FMLS (multiple and indexed vector)</a>	FEAT_SME2
0	011	<a href="#">BFVDOT</a>	FEAT_SME2
0	1x1	UNALLOCATED	-
0	100	<a href="#">SVDOT (2-way)</a>	FEAT_SME2
0	110	<a href="#">UVDOT (2-way)</a>	FEAT_SME2
1	000	<a href="#">SDOT (2-way, multiple and indexed vector)</a>	FEAT_SME2
1	001	<a href="#">FDOT (multiple and indexed vector)</a>	FEAT_SME2
1	010	<a href="#">UDOT (2-way, multiple and indexed vector)</a>	FEAT_SME2
1	011	<a href="#">BFDOT (multiple and indexed vector)</a>	FEAT_SME2
1	100	<a href="#">SDOT (4-way, multiple and indexed vector)</a>	FEAT_SME2
1	101	<a href="#">USDOT (multiple and indexed vector)</a>	FEAT_SME2
1	110	<a href="#">UDOT (4-way, multiple and indexed vector)</a>	FEAT_SME2
1	111	<a href="#">SUDOT (multiple and indexed vector)</a>	FEAT_SME2

**SME2 multi-vec indexed long MLA two sources 64-bit**

These instructions are under [SME2 Multi-vector - Indexed \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm			0	Rv		0	0	i3h	Zn			0	U	S	i3l		o1		

Decode fields U	S	Instruction Details	Feature
0	0	<a href="#">SMLALL (multiple and indexed vector)</a>	FEAT_SME_I16I64
0	1	<a href="#">SMLSLL (multiple and indexed vector)</a>	FEAT_SME_I16I64
1	0	<a href="#">UMLALL (multiple and indexed vector)</a>	FEAT_SME_I16I64
1	1	<a href="#">UMLSLL (multiple and indexed vector)</a>	FEAT_SME_I16I64

**SME2 multi-vec indexed long FMA two sources**

These instructions are under [SME2 Multi-vector - Indexed \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm			0	Rv	1	i3h	Zn			0	op	S	i3l	off2					

Decode fields op	S	Instruction Details	Feature
0	0	<a href="#">FMLAL (multiple and indexed vector)</a>	FEAT_SME2
0	1	<a href="#">FMLS (multiple and indexed vector)</a>	FEAT_SME2

Decode fields op	S	Instruction Details	Feature
1	0	<a href="#">BFMLAL (multiple and indexed vector)</a>	FEAT_SME2
1	1	<a href="#">BFMLSL (multiple and indexed vector)</a>	FEAT_SME2

**SME2 multi-vec ternary indexed two registers 64-bit**

These instructions are under [SME2 Multi-vector - Indexed \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm			0	Rv		0	0	i1	Zn			0	opc		off3				

Decode fields opc	Instruction Details	Feature
00	<a href="#">FMLA (multiple and indexed vector)</a>	FEAT_SME_F64F64
01	<a href="#">SDOT (4-way, multiple and indexed vector)</a>	FEAT_SME_I16I64
10	<a href="#">FMLS (multiple and indexed vector)</a>	FEAT_SME_F64F64
11	<a href="#">UDOT (4-way, multiple and indexed vector)</a>	FEAT_SME_I16I64

**SME2 multi-vec indexed long MLA two sources**

These instructions are under [SME2 Multi-vector - Indexed \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm			0	Rv		1	i3h	Zn			0	U	S	i3l	off2				

Decode fields U	S	Instruction Details	Feature
0	0	<a href="#">SMLAL (multiple and indexed vector)</a>	FEAT_SME2
0	1	<a href="#">SMLSL (multiple and indexed vector)</a>	FEAT_SME2
1	0	<a href="#">UMLAL (multiple and indexed vector)</a>	FEAT_SME2
1	1	<a href="#">UMLSL (multiple and indexed vector)</a>	FEAT_SME2

**SME2 Multi-vector - Indexed (Four registers)**

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Decode fields op0	op1	op2	Instruction details
00	0x	0x	<a href="#">SME2 multi-vec indexed long long MLA four sources 32-bit</a>
00	1x	0x	<a href="#">SME2 multi-vec ternary indexed four registers 16-bit</a>
01		0x	<a href="#">SME2 multi-vec ternary indexed four registers 32-bit</a>
0x		1x	UNALLOCATED
10	00	00	<a href="#">SME2 multi-vec indexed long long MLA four sources 64-bit</a>
10	00	!= 00	UNALLOCATED
10	01		UNALLOCATED
10	1x	00	<a href="#">SME2 multi-vec indexed long FMA four sources</a>
10	1x	!= 00	UNALLOCATED
11	0x	00	<a href="#">SME2 multi-vec ternary indexed four registers 64-bit</a>
11	1x	00	<a href="#">SME2 multi-vec indexed long MLA four sources</a>
11		!= 00	UNALLOCATED

**SME2 multi-vec indexed long long MLA four sources 32-bit**

These instructions are under [SME2 Multi-vector - Indexed \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				1	Rv	0	i4h	Zn			0	op	U	S	i4l	o1			

Decode fields			Instruction Details	Feature
op	U	S		
0	0	0	<a href="#">SMLALL (multiple and indexed vector)</a>	FEAT_SME2
0	0	1	<a href="#">SMLSLL (multiple and indexed vector)</a>	FEAT_SME2
0	1	0	<a href="#">UMLALL (multiple and indexed vector)</a>	FEAT_SME2
0	1	1	<a href="#">UMLSLL (multiple and indexed vector)</a>	FEAT_SME2
1		1	UNALLOCATED	-
1	0	0	<a href="#">USMLALL (multiple and indexed vector)</a>	FEAT_SME2
1	1	0	<a href="#">SUMLALL (multiple and indexed vector)</a>	FEAT_SME2

**SME2 multi-vec ternary indexed four registers 16-bit**

These instructions are under [SME2 Multi-vector - Indexed \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm				1	Rv	1	i3h	Zn			0	op	S	i3l	off3				

Decode fields		Instruction Details	Feature
op	S		
0	0	<a href="#">FMLA (multiple and indexed vector)</a>	FEAT_SME_F16F16
0	1	<a href="#">FMLS (multiple and indexed vector)</a>	FEAT_SME_F16F16
1	0	<a href="#">BFMLA (multiple and indexed vector)</a>	FEAT_SVE_B16B16
1	1	<a href="#">BFMLS (multiple and indexed vector)</a>	FEAT_SVE_B16B16

**SME2 multi-vec ternary indexed four registers 32-bit**

These instructions are under [SME2 Multi-vector - Indexed \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	0	1	Zm				1	Rv	op	i2	Zn			0	opc2		off3					

Decode fields		Instruction Details	Feature
op	opc2		
0	0x1	UNALLOCATED	-
0	000	<a href="#">FMLA (multiple and indexed vector)</a>	FEAT_SME2
0	010	<a href="#">FMLS (multiple and indexed vector)</a>	FEAT_SME2
0	100	<a href="#">SVDOT (4-way)</a>	FEAT_SME2
0	101	<a href="#">USVDOT</a>	FEAT_SME2
0	110	<a href="#">UVDOT (4-way)</a>	FEAT_SME2
0	111	<a href="#">SUVDOT</a>	FEAT_SME2
1	000	<a href="#">SDOT (2-way, multiple and indexed vector)</a>	FEAT_SME2
1	001	<a href="#">FDOT (multiple and indexed vector)</a>	FEAT_SME2
1	010	<a href="#">UDOT (2-way, multiple and indexed vector)</a>	FEAT_SME2
1	011	<a href="#">BFDOT (multiple and indexed vector)</a>	FEAT_SME2
1	100	<a href="#">SDOT (4-way, multiple and indexed vector)</a>	FEAT_SME2
1	101	<a href="#">USDOT (multiple and indexed vector)</a>	FEAT_SME2
1	110	<a href="#">UDOT (4-way, multiple and indexed vector)</a>	FEAT_SME2
1	111	<a href="#">SUDOT (multiple and indexed vector)</a>	FEAT_SME2

**SME2 multi-vec indexed long long MLA four sources 64-bit**

These instructions are under [SME2 Multi-vector - Indexed \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm			1	Rv		0	0	i3h	Zn			0	0	U	S	i3l		o1	

Decode fields		Instruction Details	Feature
U	S		
0	0	<a href="#">SMLALL (multiple and indexed vector)</a>	FEAT_SME_I16I64
0	1	<a href="#">SMLSLL (multiple and indexed vector)</a>	FEAT_SME_I16I64
1	0	<a href="#">UMLALL (multiple and indexed vector)</a>	FEAT_SME_I16I64
1	1	<a href="#">UMLSLL (multiple and indexed vector)</a>	FEAT_SME_I16I64

**SME2 multi-vec indexed long FMA four sources**

These instructions are under [SME2 Multi-vector - Indexed \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	0	1	Zm			1	Rv		1	i3h		Zn			0		0	op	S	i3l	off2	

Decode fields		Instruction Details	Feature
op	S		
0	0	<a href="#">FMLAL (multiple and indexed vector)</a>	FEAT_SME2
0	1	<a href="#">FMLS (multiple and indexed vector)</a>	FEAT_SME2
1	0	<a href="#">BFMLAL (multiple and indexed vector)</a>	FEAT_SME2
1	1	<a href="#">BFMLS (multiple and indexed vector)</a>	FEAT_SME2

**SME2 multi-vec ternary indexed four registers 64-bit**

These instructions are under [SME2 Multi-vector - Indexed \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm			1	Rv		0	op	i1	Zn			0	0	opc2		off3			

Decode fields		Instruction Details	Feature
op	opc2		
0	00	<a href="#">FMLA (multiple and indexed vector)</a>	FEAT_SME_F64F64
0	01	<a href="#">SDOT (4-way, multiple and indexed vector)</a>	FEAT_SME_I16I64
0	10	<a href="#">FMLS (multiple and indexed vector)</a>	FEAT_SME_F64F64
0	11	<a href="#">UDOT (4-way, multiple and indexed vector)</a>	FEAT_SME_I16I64
1	×0	UNALLOCATED	-
1	01	<a href="#">SVDOT (4-way)</a>	FEAT_SME_I16I64
1	11	<a href="#">UVDOT (4-way)</a>	FEAT_SME_I16I64

**SME2 multi-vec indexed long MLA four sources**

These instructions are under [SME2 Multi-vector - Indexed \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	1	Zm			1	Rv		1	i3h		Zn			0 0		U	S	i3l	off2		

Decode fields		Instruction Details	Feature
U	S		
0	0	<a href="#">SMLAL (multiple and indexed vector)</a>	FEAT_SME2
0	1	<a href="#">SMLS (multiple and indexed vector)</a>	FEAT_SME2

Decode fields		Instruction Details	Feature
U	S		
1	0	<a href="#">UMLAL (multiple and indexed vector)</a>	FEAT_SME2
1	1	<a href="#">UMLSL (multiple and indexed vector)</a>	FEAT_SME2

## SME2 Multi-vector - SVE Select

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11000001										1					op0		100								op1						op2

Decode fields			Instruction details	Feature
op0	op1	op2		
01	00	00	<a href="#">SEL — Four registers</a>	FEAT_SME2
01	00	!= 00	UNALLOCATED	-
01	!= 00		UNALLOCATED	-
11			UNALLOCATED	-
x0	x0	x0	<a href="#">SEL — Two registers</a>	FEAT_SME2
x0	x0	x1	UNALLOCATED	-
x0	x1		UNALLOCATED	-

## SME2 Multi-vector - SVE Constructive Binary

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11000001									op0	1							110			op1										op2	

Decode fields			Instruction details
op0	op1	op2	
00	101		<a href="#">SME2 multi-vec quadwords ZIP two registers</a>
01	101		UNALLOCATED
10	101		UNALLOCATED
11	101		<a href="#">SME2 multi-vec saturating shift right narrow two registers</a>
	000		<a href="#">SME2 multi-vec FCLAMP two registers</a>
	001		<a href="#">SME2 multi-vec CLAMP two registers</a>
	010	0	<a href="#">SME2 multi-vec FCLAMP four registers</a>
	011	0	<a href="#">SME2 multi-vec CLAMP four registers</a>
	01x	1	UNALLOCATED
	100		<a href="#">SME2 multi-vec ZIP two registers</a>
	11x		<a href="#">SME2 multi-vec saturating shift right narrow four registers</a>

## SME2 multi-vec quadwords ZIP two registers

These instructions are under [SME2 Multi-vector - SVE Constructive Binary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	Zm					1	1	0	1	0	1	Zn					Zd			op	

Decode fields	Instruction Details	Feature
op		
0	<a href="#">ZIP (two registers)</a>	FEAT_SME2



Decode fields op	Instruction Details	Feature
1	<a href="#">UZP (two registers)</a>	FEAT_SME2

**SME2 multi-vec saturating shift right narrow two registers**

These instructions are under [SME2 Multi-vector - SVE Constructive Binary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	op	imm4				1	1	0	1	0	1	Zn			U	Zd					

Decode fields op	U	Instruction Details	Feature
0	0	<a href="#">SQRSHR (two registers)</a>	FEAT_SME2
0	1	<a href="#">UQRSHR (two registers)</a>	FEAT_SME2
1	0	<a href="#">SQRSHRU (two registers)</a>	FEAT_SME2
1	1	UNALLOCATED	-

**SME2 multi-vec FCLAMP two registers**

These instructions are under [SME2 Multi-vector - SVE Constructive Binary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size		1	Zm				1	1	0	0	0	0	Zn			Zd			op				

Decode fields size	op	Instruction Details	Feature
	1	UNALLOCATED	-
!= 00	0	<a href="#">FCLAMP</a>	FEAT_SME2
00	0	<a href="#">BFCLAMP</a>	FEAT_SVE_B16B16

**SME2 multi-vec CLAMP two registers**

These instructions are under [SME2 Multi-vector - SVE Constructive Binary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size		1	Zm				1	1	0	0	0	1	Zn			Zd			U				

Decode fields U	Instruction Details	Feature
0	<a href="#">SCLAMP</a>	FEAT_SME2
1	<a href="#">UCLAMP</a>	FEAT_SME2

**SME2 multi-vec FCLAMP four registers**

These instructions are under [SME2 Multi-vector - SVE Constructive Binary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size		1	Zm				1	1	0	0	1	0	Zn			Zd			0	op			

Decode fields size	op	Instruction Details	Feature
	1	UNALLOCATED	-
!= 00	0	<a href="#">FCLAMP</a>	FEAT_SME2
00	0	<a href="#">BFCLAMP</a>	FEAT_SVE_B16B16

## SME2 multi-vec CLAMP four registers

These instructions are under [SME2 Multi-vector - SVE Constructive Binary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	Zm						1	1	0	0	1	1	Zn					Zd		0	U	

Decode fields U	Instruction Details	Feature
0	<a href="#">SCLAMP</a>	FEAT_SME2
1	<a href="#">UCLAMP</a>	FEAT_SME2

## SME2 multi-vec ZIP two registers

These instructions are under [SME2 Multi-vector - SVE Constructive Binary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	Zm						1	1	0	1	0	0	Zn					Zd			op	

Decode fields op	Instruction Details	Feature
0	<a href="#">ZIP (two registers)</a>	FEAT_SME2
1	<a href="#">UZP (two registers)</a>	FEAT_SME2

### SME2 multi-vec saturating shift right narrow four registers

These instructions are under [SME2 Multi-vector - SVE Constructive Binary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	tsize	1	imm5				1	1	0	1	1	N	Zn			op	U	Zd						

Decode fields			Instruction Details	Feature
N	op	U		
	1	1	UNALLOCATED	-
0	0	0	<a href="#">SQRSHR (four registers)</a>	FEAT_SME2
0	0	1	<a href="#">UQRSHR (four registers)</a>	FEAT_SME2
0	1	0	<a href="#">SQRSHRU (four registers)</a>	FEAT_SME2
1	0	0	<a href="#">SQRSHRN</a>	FEAT_SME2
1	0	1	<a href="#">UQRSHRN</a>	FEAT_SME2
1	1	0	<a href="#">SQRSHRUN</a>	FEAT_SME2

## SME2 Multi-vector - SVE Constructive Unary

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
11000001								op0		1	op1						111000									op2					op3	

Decode fields				Instruction details
op0	op1	op2	op3	
00	00001		x0	<a href="#">SME2 multi-vec FP to int convert two registers</a>
00	00001		x1	UNALLOCATED
00	00010		x0	<a href="#">SME2 multi-vec int to FP two registers</a>
00	00010		x1	UNALLOCATED
00	00100			UNALLOCATED
00	10001	0x	00	<a href="#">SME2 multi-vec FP to int convert four registers</a>

00	10001	0x	!= 00	UNALLOCATED
00	10001	1x		UNALLOCATED
00	10010	0x	00	<a href="#">SME2 multi-vec int to FP four registers</a>
00	10010	0x	!= 00	UNALLOCATED
00	10010	1x		UNALLOCATED
00	10111	00	x0	<a href="#">SME2 multi-vec quadwords ZIP four registers</a>
00	1011x	00	x1	UNALLOCATED
00	1011x	!= 00		UNALLOCATED
00	10x00			UNALLOCATED
01	00001			UNALLOCATED
01	x0010			UNALLOCATED
0x	00000			<a href="#">SME2 multi-vec FP down convert two registers</a>
0x	00011			<a href="#">SME2 multi-vec int down convert two registers</a>
0x	0011x			UNALLOCATED
10	00000			<a href="#">SME2 multi-vec convert two registers</a>
10	00001			UNALLOCATED
10	00x1x			UNALLOCATED
11	000xx			UNALLOCATED
11	0011x			UNALLOCATED
1x	10010			UNALLOCATED
!= 00	1000x			UNALLOCATED
!= 00	10110	00	x1	UNALLOCATED
!= 00	10110	01		UNALLOCATED
!= 00	10110	1x		UNALLOCATED
!= 00	10111			UNALLOCATED
!= 00	x0100			UNALLOCATED
	00101			<a href="#">SME2 multi-vec unpack two registers</a>
	01xxx	x0	x0	<a href="#">SME2 multi-vec FRINT two registers</a>
	01xxx	x0	x1	UNALLOCATED
	01xxx	x1		UNALLOCATED
	10011			<a href="#">SME2 multi-vec int down convert four registers</a>
	10101	x0	0x	<a href="#">SME2 multi-vec unpack four registers</a>
	10101	x0	1x	UNALLOCATED
	10101	x1		UNALLOCATED
	10110	00	x0	<a href="#">SME2 multi-vec ZIP four registers</a>
	11xxx	00	00	<a href="#">SME2 multi-vec FRINT four registers</a>
	11xxx	00	!= 00	UNALLOCATED
	11xxx	!= 00		UNALLOCATED

**SME2 multi-vec FP to int convert two registers**

These instructions are under [SME2 Multi-vector - SVE Constructive Unary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	1	1	1	0	0	0			Zn		U			Zd		0

Decode fields	Instruction Details	Feature
U		
0	<a href="#">FCVTZS</a>	FEAT_SME2
1	<a href="#">FCVTZU</a>	FEAT_SME2

**SME2 multi-vec int to FP two registers**

These instructions are under [SME2 Multi-vector - SVE Constructive Unary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	0	0	1	0	1	1	1	0	0	0	Zn			U	Zd			0		

Decode fields U	Instruction Details	Feature
0	<a href="#">SCVTF</a>	FEAT_SME2
1	<a href="#">UCVTF</a>	FEAT_SME2

**SME2 multi-vec FP to int convert four registers**

These instructions are under [SME2 Multi-vector - SVE Constructive Unary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	1	1	0	0	0	Zn			0	U	Zd			0	0

Decode fields U	Instruction Details	Feature
0	<a href="#">FCVTZS</a>	FEAT_SME2
1	<a href="#">FCVTZU</a>	FEAT_SME2

**SME2 multi-vec int to FP four registers**

These instructions are under [SME2 Multi-vector - SVE Constructive Unary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	0	0	1	0	1	1	1	0	0	0	Zn			0	U	Zd			0	0

Decode fields U	Instruction Details	Feature
0	<a href="#">SCVTF</a>	FEAT_SME2
1	<a href="#">UCVTF</a>	FEAT_SME2

**SME2 multi-vec quadwords ZIP four registers**

These instructions are under [SME2 Multi-vector - SVE Constructive Unary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	0	1	1	1	1	1	1	0	0	0	Zn			0	0	Zd			op	0

Decode fields op	Instruction Details	Feature
0	<a href="#">ZIP (four registers)</a>	FEAT_SME2
1	<a href="#">UZP (four registers)</a>	FEAT_SME2

**SME2 multi-vec FP down convert two registers**

These instructions are under [SME2 Multi-vector - SVE Constructive Unary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	op	1	0	0	0	0	0	1	1	1	0	0	0	Zn			N	Zd					

Decode fields op	Decode fields N	Instruction Details	Feature
0	0	<a href="#">FCVT (narrowing)</a>	FEAT_SME2

Decode fields op	N	Instruction Details	Feature
0	1	<a href="#">FCVTN</a>	FEAT_SME2
1	0	<a href="#">BFCVT</a>	FEAT_SME2
1	1	<a href="#">BFCVTN</a>	FEAT_SME2

**SME2 multi-vec int down convert two registers**

These instructions are under [SME2 Multi-vector - SVE Constructive Unary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	op	1	0	0	0	1	1	1	1	1	0	0	0	Zn			U	Zd					

Decode fields op	U	Instruction Details	Feature
0	0	<a href="#">SQCVT (two registers)</a>	FEAT_SME2
0	1	<a href="#">UQCVT (two registers)</a>	FEAT_SME2
1	0	<a href="#">SQCVTU (two registers)</a>	FEAT_SME2
1	1	UNALLOCATED	-

**SME2 multi-vec convert two registers**

These instructions are under [SME2 Multi-vector - SVE Constructive Unary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	0	0	0	0	0	1	1	1	0	0	0	Zn			Zd			L			

Decode fields L	Instruction Details	Feature
0	<a href="#">FCVT (widening)</a>	FEAT_SME_F16F16
1	<a href="#">FCVTL</a>	FEAT_SME_F16F16

**SME2 multi-vec unpack two registers**

These instructions are under [SME2 Multi-vector - SVE Constructive Unary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0	0	1	0	1	1	1	1	0	0	0	Zn			Zd			U				

Decode fields U	Instruction Details	Feature
0	<a href="#">SUNPK</a>	FEAT_SME2
1	<a href="#">UUNPK</a>	FEAT_SME2

**SME2 multi-vec FRINT two registers**

These instructions are under [SME2 Multi-vector - SVE Constructive Unary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0	1	opc		1	1	1	0	0	0	Zn			0	Zd			0				

Decode fields size	opc	Instruction Details	Feature
0x		UNALLOCATED	-
10	000	<a href="#">FRINTN</a>	FEAT_SME2
10	001	<a href="#">FRINTP</a>	FEAT_SME2

Decode fields size	opc	Instruction Details	Feature
10	010	<a href="#">FRINTM</a>	FEAT_SME2
10	011	UNALLOCATED	-
10	100	<a href="#">FRINTA</a>	FEAT_SME2
10	101	UNALLOCATED	-
10	11x	UNALLOCATED	-
11		UNALLOCATED	-

### SME2 multi-vec int down convert four registers

These instructions are under [SME2 Multi-vector - SVE Constructive Unary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	sz	op	1	1	0	0	1	1	1	1	1	0	0	0		Zn		N	U			Zd		

Decode fields op	N	U	Instruction Details	Feature
0	0	0	<a href="#">SQCVT (four registers)</a>	FEAT_SME2
0	0	1	<a href="#">UQCVT (four registers)</a>	FEAT_SME2
0	1	0	<a href="#">SQCVTN</a>	FEAT_SME2
0	1	1	<a href="#">UQCVTN</a>	FEAT_SME2
1		1	UNALLOCATED	-
1	0	0	<a href="#">SQCVTU (four registers)</a>	FEAT_SME2
1	1	0	<a href="#">SQCVTUN</a>	FEAT_SME2

### SME2 multi-vec unpack four registers

These instructions are under [SME2 Multi-vector - SVE Constructive Unary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size		1	1	0	1	0	1	1	1	1	0	0	0		Zn		0		Zd		0	U	

Decode fields U	Instruction Details	Feature
0	<a href="#">SUNPK</a>	FEAT_SME2
1	<a href="#">UUNPK</a>	FEAT_SME2

### SME2 multi-vec ZIP four registers

These instructions are under [SME2 Multi-vector - SVE Constructive Unary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size		1	1	0	1	1	0	1	1	1	0	0	0		Zn		0	0		Zd		op	0

Decode fields op	Instruction Details	Feature
0	<a href="#">ZIP (four registers)</a>	FEAT_SME2
1	<a href="#">UZP (four registers)</a>	FEAT_SME2

### SME2 multi-vec FRINT four registers

These instructions are under [SME2 Multi-vector - SVE Constructive Unary](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size			1	1	1	opc			1	1	1	0	0	0	Zn		0	0	Zd		0	0	

Decode fields size	opc	Instruction Details	Feature
0x		UNALLOCATED	-
10	000	<a href="#">FRINTN</a>	FEAT_SME2
10	001	<a href="#">FRINTP</a>	FEAT_SME2
10	010	<a href="#">FRINTM</a>	FEAT_SME2
10	011	UNALLOCATED	-
10	100	<a href="#">FRINTA</a>	FEAT_SME2
10	101	UNALLOCATED	-
10	11x	UNALLOCATED	-
11		UNALLOCATED	-

## SME2 Multi-vector - Multiple Vectors SVE Destructive (Two registers)

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11000001										1						010110					op0		op1		op2						

Decode fields			Instruction details
op0	op1	op2	
0	000	0x	<a href="#">SME2 multiple vectors int min/max two registers</a>
0	010		<a href="#">SME2 multiple vectors FP min/max two registers</a>
0	0x1		UNALLOCATED
0	10x		<a href="#">SME2 multiple vectors shift two registers</a>
0	11x		UNALLOCATED
1	000	00	<a href="#">SME2 multi-vector signed saturating doubling multiply high two registers</a>
1	000	01	UNALLOCATED
1	!= 000		UNALLOCATED
	000	1x	UNALLOCATED

## SME2 multiple vectors int min/max two registers

These instructions are under [SME2 Multi-vector - Multiple Vectors SVE Destructive \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	size			1	Zm			0	1	0	1	1	0	0	0		0	0	0	op		Zdn			U

Decode fields op	U	Instruction Details	Feature
0	0	<a href="#">SMAX (multiple vectors)</a>	FEAT_SME2
0	1	<a href="#">UMAX (multiple vectors)</a>	FEAT_SME2
1	0	<a href="#">SMIN (multiple vectors)</a>	FEAT_SME2
1	1	<a href="#">UMIN (multiple vectors)</a>	FEAT_SME2

## SME2 multiple vectors FP min/max two registers

These instructions are under [SME2 Multi-vector - Multiple Vectors SVE Destructive \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size			1	Zm			0	1	0	1	1	0	0	0 1 0		opc		Zdn			o2		

Decode fields			Instruction Details	Feature
size	opc	o2		
	1x		UNALLOCATED	-
!= 00	00	0	<a href="#">FMAX (multiple vectors)</a>	FEAT_SME2
!= 00	00	1	<a href="#">FMIN (multiple vectors)</a>	FEAT_SME2
!= 00	01	0	<a href="#">FMAXNM (multiple vectors)</a>	FEAT_SME2
!= 00	01	1	<a href="#">FMINNM (multiple vectors)</a>	FEAT_SME2
00	00	0	<a href="#">BFMAX (multiple vectors)</a>	FEAT_SVE_B16B16
00	00	1	<a href="#">BFMIN (multiple vectors)</a>	FEAT_SVE_B16B16
00	01	0	<a href="#">BFMAXNM (multiple vectors)</a>	FEAT_SVE_B16B16
00	01	1	<a href="#">BFMINNM (multiple vectors)</a>	FEAT_SVE_B16B16

### SME2 multiple vectors shift two registers

These instructions are under [SME2 Multi-vector - Multiple Vectors SVE Destructive \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1		Zm				0	1	0	1	1	0	0	1	0		opc		Zdn				U

Decode fields		Instruction Details	Feature
opc	U		
000		UNALLOCATED	-
001	0	<a href="#">SRSHL (multiple vectors)</a>	FEAT_SME2
001	1	<a href="#">URSHL (multiple vectors)</a>	FEAT_SME2
01x		UNALLOCATED	-
1xx		UNALLOCATED	-

### SME2 multi-vector signed saturating doubling multiply high two registers

These instructions are under [SME2 Multi-vector - Multiple Vectors SVE Destructive \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1		Zm				0	1	0	1	1	0	1	0	0	0	0	0		Zdn			op

Decode fields		Instruction Details	Feature
op			
0		<a href="#">SQDMULH (multiple vectors)</a>	FEAT_SME2
1		UNALLOCATED	-

### SME2 Multi-vector - Multiple Vectors SVE Destructive (Four registers)

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										1				op0					010111		op1		op2		op3					0	

Decode fields				Instruction details
op0	op1	op2	op3	
0	0	000	0x	<a href="#">SME2 multiple vectors int min/max four registers</a>
0	0	010		<a href="#">SME2 multiple vectors FP min/max four registers</a>
0	0	0x1		UNALLOCATED
0	0	10x		<a href="#">SME2 multiple vectors shift four registers</a>
0	0	11x		UNALLOCATED
0	1	000	00	<a href="#">SME2 multi-vector signed saturating doubling multiply high four registers</a>



0	1	000	01	UNALLOCATED
0	1	!= 000		UNALLOCATED
0		000	1x	UNALLOCATED
1				UNALLOCATED

**SME2 multiple vectors int min/max four registers**

These instructions are under [SME2 Multi-vector - Multiple Vectors SVE Destructive \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	size	1	Zm			0	0	1	0	1	1	1	0	0			0	0	0	op	Zdn			0	U

Decode fields op	U	Instruction Details	Feature
0	0	<a href="#">SMAX (multiple vectors)</a>	FEAT_SME2
0	1	<a href="#">UMAX (multiple vectors)</a>	FEAT_SME2
1	0	<a href="#">SMIN (multiple vectors)</a>	FEAT_SME2
1	1	<a href="#">UMIN (multiple vectors)</a>	FEAT_SME2

**SME2 multiple vectors FP min/max four registers**

These instructions are under [SME2 Multi-vector - Multiple Vectors SVE Destructive \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size		1	Zm		0	0	1	0	1	1	1	0	0	1	0	opc		Zdn		0	o2		

Decode fields size	opc	o2	Instruction Details	Feature
	1x		UNALLOCATED	-
!= 00	00	0	<a href="#">FMAX (multiple vectors)</a>	FEAT_SME2
!= 00	00	1	<a href="#">FMIN (multiple vectors)</a>	FEAT_SME2
!= 00	01	0	<a href="#">FMAXNM (multiple vectors)</a>	FEAT_SME2
!= 00	01	1	<a href="#">FMINNM (multiple vectors)</a>	FEAT_SME2
00	00	0	<a href="#">BFMAX (multiple vectors)</a>	FEAT_SVE_B16B16
00	00	1	<a href="#">BFMIN (multiple vectors)</a>	FEAT_SVE_B16B16
00	01	0	<a href="#">BFMAXNM (multiple vectors)</a>	FEAT_SVE_B16B16
00	01	1	<a href="#">BFMINNM (multiple vectors)</a>	FEAT_SVE_B16B16

**SME2 multiple vectors shift four registers**

These instructions are under [SME2 Multi-vector - Multiple Vectors SVE Destructive \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	Zm			0	0	1	0	1	1	1	0	1	0	opc			Zdn			0	U	

Decode fields opc	U	Instruction Details	Feature
000		UNALLOCATED	-
001	0	<a href="#">SRSHL (multiple vectors)</a>	FEAT_SME2
001	1	<a href="#">URSHL (multiple vectors)</a>	FEAT_SME2
01x		UNALLOCATED	-
1xx		UNALLOCATED	-

**SME2 multi-vector signed saturating doubling multiply high four registers**

These instructions are under [SME2 Multi-vector - Multiple Vectors SVE Destructive \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	Zm			0	0	1	0	1	1	1	1	0	0	0	0	0	Zdn			0	op	

Decode fields op	Instruction Details	Feature
0	<a href="#">SQDMULH (multiple vectors)</a>	FEAT_SME2
1	UNALLOCATED	-

**SME2 Multi-vector - Multiple and Single SVE Destructive (Two registers)**

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
											10										op0										

Decode fields op0	Decode fields op1	Instruction details
0	0000x	<a href="#">SME2 single-multi int min/max two registers</a>
0	0100x	<a href="#">SME2 single-multi FP min/max two registers</a>
0	0x!= 00x	UNALLOCATED
0	10xxx	<a href="#">SME2 single-multi shift two registers</a>
0	11000	<a href="#">SME2 single-multi add two registers</a>
0	11!= 000	UNALLOCATED
1	00000	<a href="#">SME2 single-multi signed saturating doubling multiply high two registers</a>
1	!= 00000	UNALLOCATED

**SME2 single-multi int min/max two registers**

These instructions are under [SME2 Multi-vector - Multiple and Single SVE Destructive \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0	Zm				1	0	1	0	0	0	0	0	0	0	0	op	Zdn			U	

Decode fields op	Decode fields U	Instruction Details	Feature
0	0	<a href="#">SMAX (multiple and single vector)</a>	FEAT_SME2
0	1	<a href="#">UMAX (multiple and single vector)</a>	FEAT_SME2
1	0	<a href="#">SMIN (multiple and single vector)</a>	FEAT_SME2
1	1	<a href="#">UMIN (multiple and single vector)</a>	FEAT_SME2

**SME2 single-multi FP min/max two registers**

These instructions are under [SME2 Multi-vector - Multiple and Single SVE Destructive \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0	Zm				1	0	1	0	0	0	0	0	1	0	0	op	Zdn			o2	

Decode fields size	Decode fields op	Decode fields o2	Instruction Details	Feature
!= 00	0	0	<a href="#">FMAX (multiple and single vector)</a>	FEAT_SME2
!= 00	0	1	<a href="#">FMIN (multiple and single vector)</a>	FEAT_SME2
!= 00	1	0	<a href="#">FMAXNM (multiple and single vector)</a>	FEAT_SME2

Decode fields size op o2	Instruction Details			Feature
!= 00	1	1	<a href="#">FMINNM (multiple and single vector)</a>	FEAT_SME2
00	0	0	<a href="#">BFMAX (multiple and single vector)</a>	FEAT_SVE_B16B16
00	0	1	<a href="#">BFMIN (multiple and single vector)</a>	FEAT_SVE_B16B16
00	1	0	<a href="#">BFMAXNM (multiple and single vector)</a>	FEAT_SVE_B16B16
00	1	1	<a href="#">BFMINNM (multiple and single vector)</a>	FEAT_SVE_B16B16

**SME2 single-multi shift two registers**

These instructions are under [SME2 Multi-vector - Multiple and Single SVE Destructive \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0		Zm	1	0	1	0	0	0	1	0	opc		Zdn	U							

Decode fields opc U	Instruction Details			Feature
000			UNALLOCATED	-
001	0		<a href="#">SRSHL (multiple and single vector)</a>	FEAT_SME2
001	1		<a href="#">URSHL (multiple and single vector)</a>	FEAT_SME2
01x			UNALLOCATED	-
1xx			UNALLOCATED	-

**SME2 single-multi add two registers**

These instructions are under [SME2 Multi-vector - Multiple and Single SVE Destructive \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0		Zm	1	0	1	0	0	0	1	1	0	0	0		Zdn	op					

Decode fields op	Instruction Details			Feature
0			<a href="#">ADD (to vector)</a>	FEAT_SME2
1			UNALLOCATED	-

**SME2 single-multi signed saturating doubling multiply high two registers**

These instructions are under [SME2 Multi-vector - Multiple and Single SVE Destructive \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0		Zm	1	0	1	0	0	1	0	0	0	0	0	0	0		Zdn	op			

Decode fields op	Instruction Details			Feature
0			<a href="#">SQDMULH (multiple and single vector)</a>	FEAT_SME2
1			UNALLOCATED	-

**SME2 Multi-vector - Multiple and Single SVE Destructive (Four registers)**

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
											10										op0										0

Decode fields

Instruction details

op0	op1	
0	0000x	<a href="#">SME2 single-multi int min/max four registers</a>
0	0100x	<a href="#">SME2 single-multi FP min/max four registers</a>
0	0x!= 00x	UNALLOCATED
0	10xxx	<a href="#">SME2 single-multi shift four registers</a>
0	11000	<a href="#">SME2 single-multi add four registers</a>
0	11!= 000	UNALLOCATED
1	00000	<a href="#">SME2 single-multi signed saturating doubling multiply high four registers</a>
1	!= 00000	UNALLOCATED

**SME2 single-multi int min/max four registers**

These instructions are under [SME2 Multi-vector - Multiple and Single SVE Destructive \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	1	0	0	0	0	0	op	Zdn		0	U	

Decode fields		Instruction Details		Feature
op	U			
0	0	<a href="#">SMAX (multiple and single vector)</a>		FEAT_SME2
0	1	<a href="#">UMAX (multiple and single vector)</a>		FEAT_SME2
1	0	<a href="#">SMIN (multiple and single vector)</a>		FEAT_SME2
1	1	<a href="#">UMIN (multiple and single vector)</a>		FEAT_SME2

**SME2 single-multi FP min/max four registers**

These instructions are under [SME2 Multi-vector - Multiple and Single SVE Destructive \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size		1	0	Zm				1	0	1	0	1	0	0	1	0	0	op	Zdn		0	o2	

Decode fields			Instruction Details		Feature
size	op	o2			
!= 00	0	0	<a href="#">FMAX (multiple and single vector)</a>		FEAT_SME2
!= 00	0	1	<a href="#">FMIN (multiple and single vector)</a>		FEAT_SME2
!= 00	1	0	<a href="#">FMAXNM (multiple and single vector)</a>		FEAT_SME2
!= 00	1	1	<a href="#">FMINNM (multiple and single vector)</a>		FEAT_SME2
00	0	0	<a href="#">BFMAX (multiple and single vector)</a>		FEAT_SVE_B16B16
00	0	1	<a href="#">BFMIN (multiple and single vector)</a>		FEAT_SVE_B16B16
00	1	0	<a href="#">BFMAXNM (multiple and single vector)</a>		FEAT_SVE_B16B16
00	1	1	<a href="#">BFMINNM (multiple and single vector)</a>		FEAT_SVE_B16B16

**SME2 single-multi shift four registers**

These instructions are under [SME2 Multi-vector - Multiple and Single SVE Destructive \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0	Zm				1	0	1	0	1	0	1	0	opc		Zdn			0	U		

Decode fields		Instruction Details		Feature
opc	U			
000		UNALLOCATED		-
001	0	<a href="#">SRSHL (multiple and single vector)</a>		FEAT_SME2
001	1	<a href="#">URSHL (multiple and single vector)</a>		FEAT_SME2

Decode fields	Instruction Details	Feature
opc	U	
01x		UNALLOCATED
1xx		UNALLOCATED

### SME2 single-multi add four registers

These instructions are under [SME2 Multi-vector - Multiple and Single SVE Destructive \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0	Zm				1	0	1	0	1	0	1	1	0	0	0	Zdn			0	op	

Decode fields	Instruction Details	Feature
op		
0	<a href="#">ADD (to vector)</a>	FEAT_SME2
1	UNALLOCATED	-

### SME2 single-multi signed saturating doubling multiply high four registers

These instructions are under [SME2 Multi-vector - Multiple and Single SVE Destructive \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	size	1	0	Zm				1	0	1	0	1	1	0	0	0	0	0	Zdn			0	op	

Decode fields	Instruction Details	Feature
op		
0	<a href="#">SQDMULH (multiple and single vector)</a>	FEAT_SME2
1	UNALLOCATED	-

### SME2 Multi-vector - Multiple and Single Array Vectors

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
110000010									op0	1	op1	0				op2				op3											

Decode fields				Instruction details
op0	op1	op2	op3	
0	0	010	xx0x	<a href="#">SME2 single-multi long FMA two sources</a>
0	0	011		<a href="#">SME2 multiple and single vector long FMA one source</a>
0	0	100		<a href="#">SME2 single-multi FP dot product two registers</a>
0	0	101	x1xx	<a href="#">SME2 single-multi mixed dot product two registers</a>
0	1	010	xx0x	<a href="#">SME2 single-multi long FMA four sources</a>
0	1	100		<a href="#">SME2 single-multi FP dot product four registers</a>
0	1	101	x1xx	<a href="#">SME2 single-multi mixed dot product four registers</a>
1	0	010	xx0x	<a href="#">SME2 single-multi long MLA two sources</a>
1	0	011		<a href="#">SME2 multiple and single vector long MLA one source</a>
1	0	101	x1xx	<a href="#">SME2 single-multi two-way dot product two registers</a>
1	1	010	xx0x	<a href="#">SME2 single-multi long MLA four sources</a>
1	1	101	x1xx	<a href="#">SME2 single-multi two-way dot product four registers</a>
1		100		UNALLOCATED
	0	000	xxx0	<a href="#">SME2 single-multi long long MLA two sources</a>
	0	001		<a href="#">SME2 multiple and single vector long long FMA one source</a>
	0	101	x0xx	<a href="#">SME2 single-multi four-way dot product two registers</a>

	0	110	0xxx	<a href="#">SME2 single-multi ternary FP two registers</a>
	0	110	1xxx	<a href="#">SME2 single-multi ternary int two registers</a>
	0	111	0xxx	<a href="#">SME2 single-multi ternary FP16 two registers</a>
	1	000	xxx0	<a href="#">SME2 single-multi long long MLA four sources</a>
	1	0x1		UNALLOCATED
	1	101	x0xx	<a href="#">SME2 single-multi four-way dot product four registers</a>
	1	110	0xxx	<a href="#">SME2 single-multi ternary FP four registers</a>
	1	110	1xxx	<a href="#">SME2 single-multi ternary int four registers</a>
	1	111	0xxx	<a href="#">SME2 single-multi ternary FP16 four registers</a>
		000	xxx1	UNALLOCATED
		010	xx1x	UNALLOCATED
		111	1xxx	UNALLOCATED

### SME2 single-multi long FMA two sources

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			0	Rv		0	1	0	Zn			op		S	0	off2			

Decode fields op	S	Instruction Details	Feature
0	0	<a href="#">FMLAL (multiple and single vector)</a>	FEAT_SME2
0	1	<a href="#">FMLSL (multiple and single vector)</a>	FEAT_SME2
1	0	<a href="#">BFMLAL (multiple and single vector)</a>	FEAT_SME2
1	1	<a href="#">BFMLSL (multiple and single vector)</a>	FEAT_SME2

### SME2 multiple and single vector long FMA one source

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			0	Rv		0	1	1	Zn			op		S	off3				

Decode fields op	S	Instruction Details	Feature
0	0	<a href="#">FMLAL (multiple and single vector)</a>	FEAT_SME2
0	1	<a href="#">FMLSL (multiple and single vector)</a>	FEAT_SME2
1	0	<a href="#">BFMLAL (multiple and single vector)</a>	FEAT_SME2
1	1	<a href="#">BFMLSL (multiple and single vector)</a>	FEAT_SME2

### SME2 single-multi FP dot product two registers

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			0	Rv		1	0	0	Zn			opc			off3				

Decode fields opc	Instruction Details	Feature
x1	UNALLOCATED	-
00	<a href="#">FDOT (multiple and single vector)</a>	FEAT_SME2
10	<a href="#">BFDOT (multiple and single vector)</a>	FEAT_SME2

**SME2 single-multi mixed dot product two registers**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	0	Zm			0	Rv		1	0	1	Zn			U	1	off3					

Decode fields U	Instruction Details	Feature
0	<a href="#">USDOT (multiple and single vector)</a>	FEAT_SME2
1	<a href="#">SUDOT (multiple and single vector)</a>	FEAT_SME2

**SME2 single-multi long FMA four sources**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm			0	Rv		0	1	0	Zn				op	S	0	off2			

Decode fields op	S	Instruction Details	Feature
0	0	<a href="#">FMLAL (multiple and single vector)</a>	FEAT_SME2
0	1	<a href="#">FMLSL (multiple and single vector)</a>	FEAT_SME2
1	0	<a href="#">BFMLAL (multiple and single vector)</a>	FEAT_SME2
1	1	<a href="#">BFMLSL (multiple and single vector)</a>	FEAT_SME2

**SME2 single-multi FP dot product four registers**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm			0	Rv		1	0	0	Zn			opc			off3				

Decode fields opc	Instruction Details	Feature
x1	UNALLOCATED	-
00	<a href="#">FDOT (multiple and single vector)</a>	FEAT_SME2
10	<a href="#">BFDOT (multiple and single vector)</a>	FEAT_SME2

**SME2 single-multi mixed dot product four registers**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	0	1	1	Zm			0	Rv		1	0	1	Zn			U	1	off3					

Decode fields U	Instruction Details	Feature
0	<a href="#">USDOT (multiple and single vector)</a>	FEAT_SME2
1	<a href="#">SUDOT (multiple and single vector)</a>	FEAT_SME2

**SME2 single-multi long MLA two sources**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm			0	Rv		0	1	0	Zn			U		S	0	off2			

Decode fields		Instruction Details	Feature
U	S		
0	0	<a href="#">SMLAL (multiple and single vector)</a>	FEAT_SME2
0	1	<a href="#">SMLSL (multiple and single vector)</a>	FEAT_SME2
1	0	<a href="#">UMLAL (multiple and single vector)</a>	FEAT_SME2
1	1	<a href="#">UMLSL (multiple and single vector)</a>	FEAT_SME2

**SME2 multiple and single vector long MLA one source**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm			0	Rv	0	1	1	Zn			U	S	off3						

Decode fields		Instruction Details	Feature
U	S		
0	0	<a href="#">SMLAL (multiple and single vector)</a>	FEAT_SME2
0	1	<a href="#">SMLSL (multiple and single vector)</a>	FEAT_SME2
1	0	<a href="#">UMLAL (multiple and single vector)</a>	FEAT_SME2
1	1	<a href="#">UMLSL (multiple and single vector)</a>	FEAT_SME2

**SME2 single-multi two-way dot product two registers**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	0	Zm			0	Rv	1	0	1	Zn			U	1	off3						

Decode fields		Instruction Details	Feature
U			
0		<a href="#">SDOT (2-way, multiple and single vector)</a>	FEAT_SME2
1		<a href="#">UDOT (2-way, multiple and single vector)</a>	FEAT_SME2

**SME2 single-multi long MLA four sources**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	1	Zm			0	Rv	0	1	0	Zn			U	S	0	off2					

Decode fields		Instruction Details	Feature
U	S		
0	0	<a href="#">SMLAL (multiple and single vector)</a>	FEAT_SME2
0	1	<a href="#">SMLSL (multiple and single vector)</a>	FEAT_SME2
1	0	<a href="#">UMLAL (multiple and single vector)</a>	FEAT_SME2
1	1	<a href="#">UMLSL (multiple and single vector)</a>	FEAT_SME2

**SME2 single-multi two-way dot product four registers**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	1	1	Zm			0	Rv	1	0	1	Zn			U	1	off3						



Decode fields	Instruction Details	Feature
<b>U</b>		
0	<a href="#">SDOT (2-way, multiple and single vector)</a>	FEAT_SME2
1	<a href="#">UDOT (2-way, multiple and single vector)</a>	FEAT_SME2

**SME2 single-multi long long MLA two sources**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm			0	Rv	0	0	0	Zn				U	S	op	0	o1			

Decode fields	Instruction Details	Feature
<b>sz U S op</b>		
	<a href="#">SMLALL (multiple and single vector)</a>	FEAT_SME2
	<a href="#">SMLSLL (multiple and single vector)</a>	FEAT_SME2
	<a href="#">UMLALL (multiple and single vector)</a>	FEAT_SME2
	<a href="#">UMLSLL (multiple and single vector)</a>	FEAT_SME2
0	UNALLOCATED	-
0	<a href="#">USMLALL (multiple and single vector)</a>	FEAT_SME2
0	<a href="#">SUMLALL (multiple and single vector)</a>	FEAT_SME2
1	UNALLOCATED	-

**SME2 multiple and single vector long long FMA one source**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm			0	Rv		0	0	1	Zn				U	S	op	off2			

Decode fields	Instruction Details	Feature
<b>sz U S op</b>		
	<a href="#">SMLALL (multiple and single vector)</a>	FEAT_SME2
	<a href="#">SMLSLL (multiple and single vector)</a>	FEAT_SME2
	<a href="#">UMLALL (multiple and single vector)</a>	FEAT_SME2
	<a href="#">UMLSLL (multiple and single vector)</a>	FEAT_SME2
0	<a href="#">USMLALL (multiple and single vector)</a>	FEAT_SME2
0	UNALLOCATED	-
0	UNALLOCATED	-
1	UNALLOCATED	-

**SME2 single-multi four-way dot product two registers**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm			0	Rv		1	0	1	Zn				U	0	off3				

Decode fields	Instruction Details	Feature
<b>U</b>		
0	<a href="#">SDOT (4-way, multiple and single vector)</a>	FEAT_SME2
1	<a href="#">UDOT (4-way, multiple and single vector)</a>	FEAT_SME2

**SME2 single-multi ternary FP two registers**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm			0	Rv	1	1	0	Zn			0	S	off3						

Decode fields S	Instruction Details	Feature
0	<a href="#">FMLA (multiple and single vector)</a>	FEAT_SME2
1	<a href="#">FMLS (multiple and single vector)</a>	FEAT_SME2

**SME2 single-multi ternary int two registers**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm			0	Rv	1	1	0	Zn			1	S	off3						

Decode fields S	Instruction Details	Feature
0	<a href="#">ADD (array results, multiple and single vector)</a>	FEAT_SME2
1	<a href="#">SUB (array results, multiple and single vector)</a>	FEAT_SME2

**SME2 single-multi ternary FP16 two registers**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	0	Zm			0	Rv		1	1	1	Zn				0	S	off3				

Decode fields sz	Decode fields S	Instruction Details	Feature
0	0	<a href="#">FMLA (multiple and single vector)</a>	FEAT_SME_F16F16
0	1	<a href="#">FMLS (multiple and single vector)</a>	FEAT_SME_F16F16
1	0	<a href="#">BFMLA (multiple and single vector)</a>	FEAT_SVE_B16B16
1	1	<a href="#">BFMLS (multiple and single vector)</a>	FEAT_SVE_B16B16

**SME2 single-multi long long MLA four sources**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm			0	Rv		0	0	0	Zn			U	S	op	0	o1			

Decode fields sz	Decode fields U	Decode fields S	Decode fields op	Instruction Details	Feature
	0	0	0	<a href="#">SMLALL (multiple and single vector)</a>	FEAT_SME2
	0	1	0	<a href="#">SMLSLL (multiple and single vector)</a>	FEAT_SME2
	1	0	0	<a href="#">UMLALL (multiple and single vector)</a>	FEAT_SME2
	1	1	0	<a href="#">UMLSLL (multiple and single vector)</a>	FEAT_SME2
0		1	1	UNALLOCATED	-
0	0	0	1	<a href="#">USMLALL (multiple and single vector)</a>	FEAT_SME2
0	1	0	1	<a href="#">SUMLALL (multiple and single vector)</a>	FEAT_SME2
1			1	UNALLOCATED	-

**SME2 single-multi four-way dot product four registers**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm			0	Rv	1	0	1	Zn				U	0	off3					

Decode fields	Instruction Details	Feature
U		
0	<a href="#">SDOT (4-way, multiple and single vector)</a>	FEAT_SME2
1	<a href="#">UDOT (4-way, multiple and single vector)</a>	FEAT_SME2

**SME2 single-multi ternary FP four registers**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm			0	Rv		1	1	0	Zn				0	S	off3				

Decode fields	Instruction Details	Feature
S		
0	<a href="#">FMLA (multiple and single vector)</a>	FEAT_SME2
1	<a href="#">FMLS (multiple and single vector)</a>	FEAT_SME2

**SME2 single-multi ternary int four registers**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm			0	Rv		1	1	0	Zn				1	S	off3				

Decode fields	Instruction Details	Feature
S		
0	<a href="#">ADD (array results, multiple and single vector)</a>	FEAT_SME2
1	<a href="#">SUB (array results, multiple and single vector)</a>	FEAT_SME2

**SME2 single-multi ternary FP16 four registers**

These instructions are under [SME2 Multi-vector - Multiple and Single Array Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	sz	1	1	Zm			0	Rv	1	1	1	Zn			0	S	off3						

Decode fields	Instruction Details	Feature
sz	S	
0	0	<a href="#">FMLA (multiple and single vector)</a>
0	1	<a href="#">FMLS (multiple and single vector)</a>
1	0	<a href="#">BFMLA (multiple and single vector)</a>
1	1	<a href="#">BFMLS (multiple and single vector)</a>

**SME2 Multi-vector - Multiple Array Vectors (Two registers)**

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
									110000011	op0	1	op1	op2	00					op3									op4			

Decode fields					Instruction details	Feature
op0	op1	op2	op3	op4		
0	00	x0	11x	1xxxx	UNALLOCATED	-
0			010	0xx0x	<a href="#">SME2 multiple vectors long FMA two sources</a>	-
0			100	1x1xx	UNALLOCATED	-
0			100	xx0xx	<a href="#">SME2 multiple vectors FP dot product two registers</a>	-
0			101	001xx	<a href="#">USDOT (multiple vectors) — Two ZA single-vectors</a>	FEAT_SME2
0			101	011xx	UNALLOCATED	-
0			101	1xxxx	UNALLOCATED	-
1	00	x0	1xx	1xxxx	UNALLOCATED	-
1	00	x1	10x	1xxxx	UNALLOCATED	-
1	!= 00		10x	1xxxx	UNALLOCATED	-
1			010	0xx0x	<a href="#">SME2 multiple vectors long MLA two sources</a>	-
1			100	0x0xx	UNALLOCATED	-
1			101	0x1xx	<a href="#">SME2 multiple vectors two-way dot product two registers</a>	-
	00	00	111	00xxx	<a href="#">SME2 multiple vectors binary FP two registers</a>	-
	00	00	111	01xxx	<a href="#">SME2 multiple vectors binary int two registers</a>	-
	00	10	111	00xxx	<a href="#">SME2 multiple vectors binary FP16 two registers</a>	-
	00	10	111	01xxx	UNALLOCATED	-
	00	x1	110	1xxxx	UNALLOCATED	-
	00	x1	111		UNALLOCATED	-
	!= 00		110	1xxxx	UNALLOCATED	-
	!= 00		111		UNALLOCATED	-
			000	0xxx0	<a href="#">SME2 multiple vectors long long MLA two sources</a>	-
			000	0xxx1	UNALLOCATED	-
			010	0xx1x	UNALLOCATED	-
			0x0	1xxxx	UNALLOCATED	-
			0x1		UNALLOCATED	-
			100	0x1xx	<a href="#">SME2 multiple vectors ternary FP16 two registers</a>	-
			101	0x0xx	<a href="#">SME2 multiple vectors four-way dot product two registers</a>	-
			110	00xxx	<a href="#">SME2 multiple vectors ternary FP two registers</a>	-
			110	01xxx	<a href="#">SME2 multiple vectors ternary int two registers</a>	-

**SME2 multiple vectors long FMA two sources**

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm		0 0		Rv		0 1 0		Zn			0 op		S		0		off2			

Decode fields		Instruction Details	Feature
op	S		
0	0	<a href="#">FMLAL (multiple vectors)</a>	FEAT_SME2
0	1	<a href="#">FMLS� (multiple vectors)</a>	FEAT_SME2
1	0	<a href="#">BFMLAL (multiple vectors)</a>	FEAT_SME2
1	1	<a href="#">BFMLS� (multiple vectors)</a>	FEAT_SME2

**SME2 multiple vectors FP dot product two registers**

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm			0	0	Rv		1	0	0	Zn			opc		0	off3				

Decode fields opc	Instruction Details	Feature
00	<a href="#">FDOT (multiple vectors)</a>	FEAT_SME2
01	<a href="#">BFDOT (multiple vectors)</a>	FEAT_SME2
1x	UNALLOCATED	-

**SME2 multiple vectors long MLA two sources**

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	Zm			0	0	Rv		0	1	0	Zn			0	U	S	0	off2			

Decode fields U	S	Instruction Details	Feature
0	0	<a href="#">SMLAL (multiple vectors)</a>	FEAT_SME2
0	1	<a href="#">SMLSL (multiple vectors)</a>	FEAT_SME2
1	0	<a href="#">UMLAL (multiple vectors)</a>	FEAT_SME2
1	1	<a href="#">UMLSL (multiple vectors)</a>	FEAT_SME2

**SME2 multiple vectors two-way dot product two registers**

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	Zm			0	0	Rv		1	0	1	Zn			0	U	1	off3				

Decode fields U	Instruction Details	Feature
0	<a href="#">SDOT (2-way, multiple vectors)</a>	FEAT_SME2
1	<a href="#">UDOT (2-way, multiple vectors)</a>	FEAT_SME2

**SME2 multiple vectors binary FP two registers**

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	0	0		Rv		1	1	1		Zm		0	0	S		off3	

Decode fields S	Instruction Details	Feature
0	<a href="#">FADD</a>	FEAT_SME2
1	<a href="#">FSUB</a>	FEAT_SME2

**SME2 multiple vectors binary int two registers**

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	0	0		Rv		1	1	1		Zm		0	1	S		off3	

Decode fields S	Instruction Details	Feature
0	<a href="#">ADD (array accumulators)</a>	FEAT_SME2
1	<a href="#">SUB (array accumulators)</a>	FEAT_SME2

**SME2 multiple vectors binary FP16 two registers**

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	0	0	1	0	0	0	Rv	1	1	1	Zm			0		0	S	off3			

Decode fields sz	Decode fields S	Instruction Details	Feature
0	0	<a href="#">FADD</a>	FEAT_SME_F16F16
0	1	<a href="#">FSUB</a>	FEAT_SME_F16F16
1	0	<a href="#">BFADD</a>	FEAT_SVE_B16B16
1	1	<a href="#">BFSUB</a>	FEAT_SVE_B16B16

**SME2 multiple vectors long long MLA two sources**

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm			0	0	Rv	0	0	0	Zn			0	U	S	op	0	o1			

Decode fields sz	Decode fields U	Decode fields S	Decode fields op	Instruction Details	Feature
	0	0	0	<a href="#">SMLALL (multiple vectors)</a>	FEAT_SME2
	0	1	0	<a href="#">SMLSLL (multiple vectors)</a>	FEAT_SME2
	1	0	0	<a href="#">UMLALL (multiple vectors)</a>	FEAT_SME2
	1	1	0	<a href="#">UMLSLL (multiple vectors)</a>	FEAT_SME2
0	0	0	1	<a href="#">USMLALL (multiple vectors)</a>	FEAT_SME2
0	0	1	1	UNALLOCATED	-
0	1		1	UNALLOCATED	-
1			1	UNALLOCATED	-

**SME2 multiple vectors ternary FP16 two registers**

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm		0 0		Rv		1 0 0		Zn			0		S		1		off3			

Decode fields sz	Decode fields S	Instruction Details	Feature
0	0	<a href="#">FMLA (multiple vectors)</a>	FEAT_SME_F16F16
0	1	<a href="#">FMLS (multiple vectors)</a>	FEAT_SME_F16F16
1	0	<a href="#">BFMLA (multiple vectors)</a>	FEAT_SVE_B16B16
1	1	<a href="#">BFMLS (multiple vectors)</a>	FEAT_SVE_B16B16

**SME2 multiple vectors four-way dot product two registers**

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm				0	0	Rv	1	0	1	Zn				0	U	0	off3			

Decode fields	Instruction Details	Feature
U		
0	<a href="#">SDOT (4-way, multiple vectors)</a>	FEAT_SME2
1	<a href="#">UDOT (4-way, multiple vectors)</a>	FEAT_SME2

### SME2 multiple vectors ternary FP two registers

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm				0	0	Rv	1	1	0	Zn				0	0	S	off3			

Decode fields	Instruction Details	Feature
S		
0	<a href="#">FMLA (multiple vectors)</a>	FEAT_SME2
1	<a href="#">FMLS (multiple vectors)</a>	FEAT_SME2

### SME2 multiple vectors ternary int two registers

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Two registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1		Zm			0	0	Rv		1	1	0		Zn		0	1	S		off3		

Decode fields	Instruction Details	Feature
S		
0	<a href="#">ADD (array results, multiple vectors)</a>	FEAT_SME2
1	<a href="#">SUB (array results, multiple vectors)</a>	FEAT_SME2

### SME2 Multi-vector - Multiple Array Vectors (Four registers)

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
									op0	1	op1	op2			10				op3						op4			op5			

op0	op1	Decode fields	op2	op3	op4	op5	Instruction details	Feature
0	00	x0	11x	01			UNALLOCATED	-
0		x0	010	00	xx0x		<a href="#">SME2 multiple vectors long FMA four sources</a>	-
0		x0	100	01	x1xx		UNALLOCATED	-
0		x0	100	0x	x0xx		<a href="#">SME2 multiple vectors FP dot product four registers</a>	-
0		x0	101	00	01xx		<a href="#">USDOT (multiple vectors) — Four ZA single-vectors</a>	FEAT_SME2
0		x0	101	00	11xx		UNALLOCATED	-
0		x0	101	01			UNALLOCATED	-
1	00	x0	1xx	01			UNALLOCATED	-
1	!= 00	x0	10x	01			UNALLOCATED	-
1		x0	010	00	xx0x		<a href="#">SME2 multiple vectors long MLA four sources</a>	-
1		x0	100	00	x0xx		UNALLOCATED	-

1		x0	101	00	x1xx	<a href="#">SME2 multiple vectors two-way dot product four registers</a>	-
	00	00	111	00	0xxx	<a href="#">SME2 multiple vectors binary FP four registers</a>	-
	00	00	111	00	1xxx	<a href="#">SME2 multiple vectors binary int four registers</a>	-
	00	10	111	00	0xxx	<a href="#">SME2 multiple vectors binary FP16 four registers</a>	-
	00	10	111	00	1xxx	UNALLOCATED	-
	00	x0	1xx	1x		UNALLOCATED	-
	!= 00	x0	10x	1x		UNALLOCATED	-
	!= 00	x0	110	01		UNALLOCATED	-
	!= 00	x0	110	1x		UNALLOCATED	-
	!= 00	x0	111			UNALLOCATED	-
		x0	000	00	xxx0	<a href="#">SME2 multiple vectors long long MLA four sources</a>	-
		x0	000	00	xxx1	UNALLOCATED	-
		x0	010	00	xx1x	UNALLOCATED	-
		x0	0x0	!= 00		UNALLOCATED	-
		x0	0x1			UNALLOCATED	-
		x0	100	00	x1xx	<a href="#">SME2 multiple vectors ternary FP16 four registers</a>	-
		x0	101	00	x0xx	<a href="#">SME2 multiple vectors four-way dot product four registers</a>	-
		x0	110	00	0xxx	<a href="#">SME2 multiple vectors ternary FP four registers</a>	-
		x0	110	00	1xxx	<a href="#">SME2 multiple vectors ternary int four registers</a>	-
		x1				UNALLOCATED	-

### SME2 multiple vectors long FMA four sources

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1		Zm		0	1	0		Rv		0	1	0		Zn		0	0	op	S	0	off2

Decode fields op	S	Instruction Details	Feature
0	0	<a href="#">FMLAL (multiple vectors)</a>	FEAT_SME2
0	1	<a href="#">FMLS� (multiple vectors)</a>	FEAT_SME2
1	0	<a href="#">BFMLAL (multiple vectors)</a>	FEAT_SME2
1	1	<a href="#">BFMLS� (multiple vectors)</a>	FEAT_SME2

### SME2 multiple vectors FP dot product four registers

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	0	1	Zm		0		1	0	Rv		1		0	0	Zn		0		opc		0	off3	



Decode fields opc	Instruction Details	Feature
00	<a href="#">FDOT (multiple vectors)</a>	FEAT_SME2
01	<a href="#">BFDOT (multiple vectors)</a>	FEAT_SME2
1x	UNALLOCATED	-

### SME2 multiple vectors long MLA four sources

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	Zm	0	1	0	Rv	0	1	0	Zn	0	0	U	S	0	off2						

Decode fields U	S	Instruction Details	Feature
0	0	<a href="#">SMLAL (multiple vectors)</a>	FEAT_SME2
0	1	<a href="#">SMLSL (multiple vectors)</a>	FEAT_SME2
1	0	<a href="#">UMLAL (multiple vectors)</a>	FEAT_SME2
1	1	<a href="#">UMLSL (multiple vectors)</a>	FEAT_SME2

### SME2 multiple vectors two-way dot product four registers

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	Zm	0	1	0	Rv	1	0	1	Zn	0	0	U	1	off3							

Decode fields U	Instruction Details	Feature
0	<a href="#">SDOT (2-way, multiple vectors)</a>	FEAT_SME2
1	<a href="#">UDOT (2-way, multiple vectors)</a>	FEAT_SME2

### SME2 multiple vectors binary FP four registers

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	1	0	Rv	1	1	1	Zm	0	0	0	S	off3					

Decode fields S	Instruction Details	Feature
0	<a href="#">FADD</a>	FEAT_SME2
1	<a href="#">FSUB</a>	FEAT_SME2

### SME2 multiple vectors binary int four registers

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	0	0	0	0	1	0	Rv	1	1	1	Zm	0	0	1	S	off3					

Decode fields S	Instruction Details	Feature
0	<a href="#">ADD (array accumulators)</a>	FEAT_SME2
1	<a href="#">SUB (array accumulators)</a>	FEAT_SME2

**SME2 multiple vectors binary FP16 four registers**

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	0	0	1	0	1	0	Rv	1	1	1	Zm	0	0	0	S	off3					

Decode fields sz	S	Instruction Details	Feature
0	0	<a href="#">FADD</a>	FEAT_SME_F16F16
0	1	<a href="#">FSUB</a>	FEAT_SME_F16F16
1	0	<a href="#">BFADD</a>	FEAT_SVE_B16B16
1	1	<a href="#">BFSUB</a>	FEAT_SVE_B16B16

**SME2 multiple vectors long long MLA four sources**

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm	0	1	0	Rv	0	0	0	Zn	0	0	U	S	op	0	0	1				

Decode fields sz	U	S	op	Instruction Details	Feature
	0	0	0	<a href="#">SMLALL (multiple vectors)</a>	FEAT_SME2
	0	1	0	<a href="#">SMLSLL (multiple vectors)</a>	FEAT_SME2
	1	0	0	<a href="#">UMLALL (multiple vectors)</a>	FEAT_SME2
	1	1	0	<a href="#">UMLSLL (multiple vectors)</a>	FEAT_SME2
0	0	0	1	<a href="#">USMLALL (multiple vectors)</a>	FEAT_SME2
0	0	1	1	UNALLOCATED	-
0	1		1	UNALLOCATED	-
1			1	UNALLOCATED	-

**SME2 multiple vectors ternary FP16 four registers**

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm	0	1	0	Rv	1	0	0	Zn	0	0	S	1	off3							

Decode fields sz	S	Instruction Details	Feature
0	0	<a href="#">FMLA (multiple vectors)</a>	FEAT_SME_F16F16
0	1	<a href="#">FMLS (multiple vectors)</a>	FEAT_SME_F16F16
1	0	<a href="#">BFMLA (multiple vectors)</a>	FEAT_SVE_B16B16
1	1	<a href="#">BFMLS (multiple vectors)</a>	FEAT_SVE_B16B16

**SME2 multiple vectors four-way dot product four registers**

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm	0	1	0	Rv	1	0	1	Zn	0	0	U	0	off3							

Decode fields U	Instruction Details	Feature
0	<a href="#">SDOT (4-way, multiple vectors)</a>	FEAT_SME2

Decode fields U	Instruction Details	Feature
1	<a href="#">UDOT (4-way, multiple vectors)</a>	FEAT_SME2

## SME2 multiple vectors ternary FP four registers

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	0	0	1	1	sz	1	Zm		0		1	0	Rv		1		1	0	Zn		0			0	0	S	off3	

Decode fields S	Instruction Details	Feature
0	<a href="#">FMLA (multiple vectors)</a>	FEAT_SME2
1	<a href="#">FMLS (multiple vectors)</a>	FEAT_SME2

## SME2 multiple vectors ternary int four registers

These instructions are under [SME2 Multi-vector - Multiple Array Vectors \(Four registers\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	sz	1	Zm		0		1	0	Rv		1		1	0	Zn		0		0	1	S	off3	

Decode fields S	Instruction Details	Feature
0	<a href="#">ADD (array results, multiple vectors)</a>	FEAT_SME2
1	<a href="#">SUB (array results, multiple vectors)</a>	FEAT_SME2

## SME Memory

These instructions are under [SME encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110000								op0				op1				op2		op3								op4					

Decode fields					Instruction details	Feature
op0	op1	op2	op3	op4		
0xx0				0xx	<a href="#">SME load array vector (elements)</a>	-
0xx1				0xx	<a href="#">SME store array vector (elements)</a>	-
0xxx				1xx	UNALLOCATED	-
100x	00000	0	xx000	0xx	<a href="#">SME save and restore array</a>	-
100x	00000	0	xx000	1xx	UNALLOCATED	-
100x	00000	0	xx!= 000		UNALLOCATED	-
100x	!= 00000	0			UNALLOCATED	-
100x		1	00000	000	<a href="#">SME2 lookup table load/store</a>	-
100x		1	00000	!= 000	UNALLOCATED	-
100x		1	!= 00000		UNALLOCATED	-
101x					UNALLOCATED	-
110x					UNALLOCATED	-
1110				0xx	<a href="#">LD1Q</a>	FEAT_SME
1111				0xx	<a href="#">ST1Q</a>	FEAT_SME
111x				1xx	UNALLOCATED	-

**SME load array vector (elements)**

These instructions are under [SME Memory](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	0	msz	0	Rm			V	Rs	Pg		Rn			0	opc										

Decode fields msz	Instruction Details	Feature
00	<a href="#">LD1B (scalar plus scalar, tile slice)</a>	FEAT_SME
01	<a href="#">LD1H (scalar plus scalar, tile slice)</a>	FEAT_SME
10	<a href="#">LD1W (scalar plus scalar, tile slice)</a>	FEAT_SME
11	<a href="#">LD1D (scalar plus scalar, tile slice)</a>	FEAT_SME

**SME store array vector (elements)**

These instructions are under [SME Memory](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	0	msz	1	Rm			V	Rs	Pg		Rn			0	opc										

Decode fields msz	Instruction Details	Feature
00	<a href="#">ST1B (scalar plus scalar, tile slice)</a>	FEAT_SME
01	<a href="#">ST1H (scalar plus scalar, tile slice)</a>	FEAT_SME
10	<a href="#">ST1W (scalar plus scalar, tile slice)</a>	FEAT_SME
11	<a href="#">ST1D (scalar plus scalar, tile slice)</a>	FEAT_SME

**SME save and restore array**

These instructions are under [SME Memory](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	1	0	0	op	0	0	0	0	0	0		Rv		0	0	0			Rn		0			imm4	

Decode fields op	Instruction Details	Feature
0	<a href="#">LDR (vector)</a>	FEAT_SME
1	<a href="#">STR (vector)</a>	FEAT_SME

**SME2 lookup table load/store**

These instructions are under [SME Memory](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	1	0	0																	Rn		0	0	0	opc2

Decode fields opc	Decode fields opc2	Instruction Details	Feature
x0xxxx		UNALLOCATED	-
x10xxx		UNALLOCATED	-
x110xx		UNALLOCATED	-
x1110x		UNALLOCATED	-
x11110		UNALLOCATED	-
x11111	01	UNALLOCATED	-
x11111	1x	UNALLOCATED	-

Decode fields		Instruction Details	Feature
opc	opc2		
011111	00	<a href="#">LDR (ZT0)</a>	FEAT_SME2
111111	00	<a href="#">STR (ZT0)</a>	FEAT_SME2

## SVE encodings

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0				0010				op1								op2								op3							

Decode fields				Instruction details	Feature
op0	op1	op2	op3		
000	0xx0xxxx	x1xxxx		<a href="#">SVE Integer Multiply-Add - Predicated</a>	-
000	0xx0xxxx	000xxx		<a href="#">SVE Integer Binary Arithmetic - Predicated</a>	-
000	0xx0xxxx	001xxx		<a href="#">SVE Integer Reduction</a>	-
000	0xx0xxxx	100xxx		<a href="#">SVE Bitwise Shift - Predicated</a>	-
000	0xx0xxxx	101xxx		<a href="#">SVE Integer Unary Arithmetic - Predicated</a>	-
000	0xx1xxxx	000xxx		<a href="#">SVE integer add/subtract vectors (unpredicated)</a>	-
000	0xx1xxxx	001xxx		<a href="#">SVE Bitwise Logical - Unpredicated</a>	-
000	0xx1xxxx	0100xx		<a href="#">SVE Index Generation</a>	-
000	0xx1xxxx	0101xx		<a href="#">SVE Stack Allocation</a>	-
000	0xx1xxxx	011xxx		<a href="#">SVE2 Integer Multiply - Unpredicated</a>	-
000	0xx1xxxx	100xxx		<a href="#">SVE Bitwise Shift - Unpredicated</a>	-
000	0xx1xxxx	1010xx		<a href="#">SVE address generation</a>	-
000	0xx1xxxx	1011xx		<a href="#">SVE Integer Misc - Unpredicated</a>	-
000	0xx1xxxx	11xxxx		<a href="#">SVE Element Count</a>	-
000	1xx00xxx			<a href="#">SVE Bitwise Immediate</a>	-
000	1xx01xxx			<a href="#">SVE Integer Wide Immediate - Predicated</a>	-
000	1xx1xxxx	001000		<a href="#">DUP (indexed)</a>	-
000	1xx1xxxx	001001		<a href="#">SVE Permute Vector - One Source Quadwords</a>	-
000	1xx1xxxx	00101x		<a href="#">SVE table lookup (three sources)</a>	-
000	1xx1xxxx	001100		<a href="#">TBL — SVE</a>	-
000	1xx1xxxx	001101		<a href="#">TBXQ</a>	FEAT_SVE2p1
000	1xx1xxxx	001110		<a href="#">SVE Permute Vector - Unpredicated</a>	-
000	1xx1xxxx	001111		UNALLOCATED	-
000	1xx1xxxx	010xxx		<a href="#">SVE Permute Predicate</a>	-
000	1xx1xxxx	011xxx		<a href="#">SVE permute vector elements</a>	-
000	1xx1xxxx	10xxxx		<a href="#">SVE Permute Vector - Predicated</a>	-
000	1xx1xxxx	11xxxx		<a href="#">SEL (vectors)</a>	-
000	10x1xxxx	000xxx		<a href="#">SVE Permute Vector - Extract</a>	-
000	11x1xxxx	000xxx		<a href="#">SVE Permute Vector - Segments</a>	-
001	0xx0xxxx			<a href="#">SVE Integer Compare - Vectors</a>	-
001	0xx1xxxx			<a href="#">SVE integer compare with unsigned immediate</a>	-
001	1xx0xxxx	x0xxxx		<a href="#">SVE integer compare with signed immediate</a>	-
001	1xx00xxx	01xxxx		<a href="#">SVE predicate logical operations</a>	-
001	1xx00xxx	11xxxx		<a href="#">SVE Propagate Break</a>	-
001	1xx01xxx	01xxxx		<a href="#">SVE Partition Break</a>	-
001	1xx01xxx	11xxxx		<a href="#">SVE Predicate Misc</a>	-

001	1xx1xxxx	00xxxx		<a href="#">SVE Integer Compare - Scalars</a>	-
001	1xx1xxxx	01xxxx	0	<a href="#">SVE broadcast predicate element</a>	-
001	1xx1xxxx	01xxxx	1	<a href="#">SVE Scalar Integer Compare - Predicate-as-counter</a>	-
001	1xx1xxxx	11xxxx		<a href="#">SVE Integer Wide Immediate - Unpredicated</a>	-
001	1xx100xx	10xxxx		<a href="#">SVE Predicate Count</a>	-
001	1xx101xx	1000xx		<a href="#">SVE Inc/Dec by Predicate Count</a>	-
001	1xx101xx	1001xx		<a href="#">SVE Write FFR</a>	-
001	1xx101xx	101xxx		UNALLOCATED	-
001	1xx11xxx	10xxxx		UNALLOCATED	-
010	0xx0xxxx	0xxxxx		<a href="#">SVE Integer Multiply-Add - Unpredicated</a>	-
010	0xx0xxxx	10xxxx		<a href="#">SVE2 Integer - Predicated</a>	-
010	0xx0xxxx	11000x		<a href="#">SVE integer clamp</a>	-
010	0xx0xxxx	1101xx		UNALLOCATED	-
010	0xx0xxxx	111xxx		<a href="#">SVE permute vector elements (quadwords)</a>	-
010	0xx1xxxx			<a href="#">SVE Multiply - Indexed</a>	-
010	0x10xxxx	11001x		UNALLOCATED	-
010	0000xxxx	11001x		<a href="#">SVE two-way dot product</a>	-
010	0100xxxx	11001x		<a href="#">SVE two-way dot product (indexed)</a>	-
010	1xx0xxxx	0xxxxx		<a href="#">SVE2 Widening Integer Arithmetic</a>	-
010	1xx0xxxx	10xxxx		<a href="#">SVE Misc</a>	-
010	1xx0xxxx	11xxxx		<a href="#">SVE2 Accumulate</a>	-
010	1xx1xxxx	0xxxxx		<a href="#">SVE2 Narrowing</a>	-
010	1xx1xxxx	100xxx		<a href="#">SVE2 character match</a>	-
010	1xx1xxxx	101xxx		<a href="#">SVE2 Histogram Computation - Segment</a>	-
010	1xx1xxxx	110xxx		<a href="#">HISTCNT</a>	-
010	1xx1xxxx	111xxx		<a href="#">SVE2 Crypto Extensions</a>	-
011	0xx0xxxx	0xxxxx		<a href="#">FCMLA (vectors)</a>	-
011	0xx00x1x	1xxxxx		UNALLOCATED	-
011	0xx00000	100xxx		<a href="#">FCADD</a>	-
011	0xx00000	101xxx		UNALLOCATED	-
011	0xx00000	11xxxx		UNALLOCATED	-
011	0xx00001	1xxxxx		UNALLOCATED	-
011	0xx0010x	100xxx		UNALLOCATED	-
011	0xx0010x	101xxx		<a href="#">SVE floating-point convert precision odd elements</a>	-
011	0xx0010x	11xxxx		UNALLOCATED	-
011	0xx010xx	100xxx		<a href="#">SVE2 floating-point pairwise operations</a>	-
011	0xx010xx	101xxx		<a href="#">SVE floating-point recursive reduction (quadwords)</a>	-
011	0xx010xx	11xxxx		UNALLOCATED	-
011	0xx011xx	1xxxxx		UNALLOCATED	-
011	0xx1xxxx	0000xx		<a href="#">SVE floating-point multiply-add (indexed)</a>	-
011	0xx1xxxx	0001xx		<a href="#">SVE floating-point complex multiply-add (indexed)</a>	-
011	0xx1xxxx	0010x0		<a href="#">SVE floating-point multiply (indexed)</a>	-
011	0xx1xxxx	001001		<a href="#">SVE FP clamp</a>	-
011	0xx1xxxx	001011		UNALLOCATED	-
011	0xx1xxxx	0011xx		UNALLOCATED	-
011	0xx1xxxx	01x0xx		<a href="#">SVE Floating Point Widening Multiply-Add - Indexed</a>	-
011	0xx1xxxx	01x1xx		UNALLOCATED	-
011	0xx1xxxx	10x00x		<a href="#">SVE Floating Point Widening Multiply-Add</a>	-
011	0xx1xxxx	10x01x		UNALLOCATED	-

011	0xx1xxxx	10x1xx		UNALLOCATED	-
011	0xx1xxxx	110xxx		UNALLOCATED	-
011	0xx1xxxx	111000		UNALLOCATED	-
011	0xx1xxxx	111001		<a href="#">SVE floating point matrix multiply accumulate</a>	-
011	0xx1xxxx	11101x		UNALLOCATED	-
011	0xx1xxxx	1111xx		UNALLOCATED	-
011	1xx0xxxx	x1xxxx		<a href="#">SVE floating-point compare vectors</a>	-
011	1xx0xxxx	000xxx		<a href="#">SVE floating-point arithmetic (unpredicated)</a>	-
011	1xx0xxxx	100xxx		<a href="#">SVE Floating Point Arithmetic - Predicated</a>	-
011	1xx0xxxx	101xxx		<a href="#">SVE Floating Point Unary Operations - Predicated</a>	-
011	1xx000xx	001xxx		<a href="#">SVE floating-point recursive reduction</a>	-
011	1xx001xx	0010xx		UNALLOCATED	-
011	1xx001xx	0011xx		<a href="#">SVE Floating Point Unary Operations - Unpredicated</a>	-
011	1xx010xx	001xxx		<a href="#">SVE Floating Point Compare - with Zero</a>	-
011	1xx011xx	001xxx		<a href="#">SVE Floating Point Accumulating Reduction</a>	-
011	1xx1xxxx			<a href="#">SVE Floating Point Multiply-Add</a>	-
100				<a href="#">SVE Memory - 32-bit Gather and Unsized Contiguous</a>	-
101				<a href="#">SVE Memory - Contiguous Load</a>	-
110				<a href="#">SVE Memory - 64-bit Gather</a>	-
111		0x0xxx		<a href="#">SVE Memory - Contiguous Store and Unsized Contiguous</a>	-
111		001xxx		<a href="#">SVE Memory - Non-temporal and Quadword Scatter Store</a>	-
111		011xxx		<a href="#">SVE Memory - Non-temporal and Multi-register Contiguous Store</a>	-
111		1x0xxx		<a href="#">SVE Memory - Scatter with Optional Sign Extend</a>	-
111		101xxx		<a href="#">SVE Memory - Scatter</a>	-
111		111xxx		<a href="#">SVE Memory - Contiguous Store with Immediate Offset</a>	-

## SVE Integer Multiply-Add - Predicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000100									0							op0	1														

### Decode fields

op0

### Instruction details

0	<a href="#">SVE integer multiply-accumulate writing addend (predicated)</a>
1	<a href="#">SVE integer multiply-add writing multiplicand (predicated)</a>

## SVE integer multiply-accumulate writing addend (predicated)

These instructions are under [SVE Integer Multiply-Add - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size		0	Zm				0	1	op	Pg		Zn				Zda							

### Decode fields

op

### Instruction Details

0	<a href="#">MLA (vectors)</a>
1	<a href="#">MLS (vectors)</a>

**SVE integer multiply-add writing multiplicand (predicated)**

These instructions are under [SVE Integer Multiply-Add - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0				Zm			1	1	op		Pg				Za					Zdn		

Decode fields op	Instruction Details
0	<a href="#">MAD</a>
1	<a href="#">MSB</a>

**SVE Integer Binary Arithmetic - Predicated**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					00000100					0		op0						000													

Decode fields op0	Instruction details
00x	<a href="#">SVE integer add/subtract vectors (predicated)</a>
01x	<a href="#">SVE integer min/max/difference (predicated)</a>
100	<a href="#">SVE integer multiply vectors (predicated)</a>
101	<a href="#">SVE integer divide vectors (predicated)</a>
11x	<a href="#">SVE bitwise logical operations (predicated)</a>

**SVE integer add/subtract vectors (predicated)**

These instructions are under [SVE Integer Binary Arithmetic - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	0	0		opc			0	0	0		Pg				Zm					Zdn		

Decode fields opc	Instruction Details
000	<a href="#">ADD (vectors, predicated)</a>
001	<a href="#">SUB (vectors, predicated)</a>
010	UNALLOCATED
011	<a href="#">SUBR (vectors)</a>
1xx	UNALLOCATED

**SVE integer min/max/difference (predicated)**

These instructions are under [SVE Integer Binary Arithmetic - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	0	1		opc	U	0	0	0		Pg				Zm						Zdn		

Decode fields opc	U	Instruction Details
00	0	<a href="#">SMAX (vectors)</a>
00	1	<a href="#">UMAX (vectors)</a>
01	0	<a href="#">SMIN (vectors)</a>
01	1	<a href="#">UMIN (vectors)</a>
10	0	<a href="#">SABD</a>



Decode fields opc	U	Instruction Details
10	1	<a href="#">UABD</a>
11		UNALLOCATED

**SVE integer multiply vectors (predicated)**

These instructions are under [SVE Integer Binary Arithmetic - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size		0	1	0	0	H	U	0	0	0	Pg						Zm				Zdn		

Decode fields H	U	Instruction Details
0	0	<a href="#">MUL (vectors, predicated)</a>
0	1	UNALLOCATED
1	0	<a href="#">SMULH (predicated)</a>
1	1	<a href="#">UMULH (predicated)</a>

**SVE integer divide vectors (predicated)**

These instructions are under [SVE Integer Binary Arithmetic - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size		0	1	0	1	R	U	0	0	0	Pg						Zm				Zdn		

Decode fields R	U	Instruction Details
0	0	<a href="#">SDIV</a>
0	1	<a href="#">UDIV</a>
1	0	<a href="#">SDIVR</a>
1	1	<a href="#">UDIVR</a>

**SVE bitwise logical operations (predicated)**

These instructions are under [SVE Integer Binary Arithmetic - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size		0	1	1	opc			0	0	0	Pg						Zm				Zdn		

Decode fields opc	Instruction Details
000	<a href="#">ORR (vectors, predicated)</a>
001	<a href="#">EOR (vectors, predicated)</a>
010	<a href="#">AND (vectors, predicated)</a>
011	<a href="#">BIC (vectors, predicated)</a>
1xx	UNALLOCATED

**SVE Integer Reduction**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										0	op0								001												

Decode fields op0	Instruction details
000	<a href="#">SVE integer add reduction (predicated)</a>
001	<a href="#">SVE integer add reduction (quadwords)</a>
010	<a href="#">SVE integer min/max reduction (predicated)</a>
011	<a href="#">SVE integer min/max reduction (quadwords)</a>
10x	<a href="#">SVE constructive prefix (predicated)</a>
110	<a href="#">SVE bitwise logical reduction (predicated)</a>
111	<a href="#">SVE bitwise logical reduction (quadwords)</a>

**SVE integer add reduction (predicated)**

These instructions are under [SVE Integer Reduction](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	0	0	0	op	U	0	0	1	Pg	Zn				Vd								

Decode fields op	U	Instruction Details
0	0	<a href="#">SADDV</a>
0	1	<a href="#">UADDV</a>
1		UNALLOCATED

**SVE integer add reduction (quadwords)**

These instructions are under [SVE Integer Reduction](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	0	0	1	op	U	0	0	1	Pg	Zn			Vd									

Decode fields op	U	Instruction Details	Feature
0	0	UNALLOCATED	-
0	1	<a href="#">ADDQV</a>	FEAT_SVE2p1
1		UNALLOCATED	-

**SVE integer min/max reduction (predicated)**

These instructions are under [SVE Integer Reduction](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	0	1	0	op	U	0	0	1	Pg	Zn			Vd									

Decode fields op	U	Instruction Details
0	0	<a href="#">SMAV</a>
0	1	<a href="#">UMAV</a>
1	0	<a href="#">SMINV</a>
1	1	<a href="#">UMINV</a>

**SVE integer min/max reduction (quadwords)**

These instructions are under [SVE Integer Reduction](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	0	1	1	op	U	0	0	1	Pg	Zn				Vd								

Decode fields op	U	Instruction Details	Feature
0	0	<a href="#">SMAXQV</a>	FEAT_SVE2p1
0	1	<a href="#">UMAXQV</a>	FEAT_SVE2p1
1	0	<a href="#">SMINQV</a>	FEAT_SVE2p1
1	1	<a href="#">UMINQV</a>	FEAT_SVE2p1

**SVE constructive prefix (predicated)**

These instructions are under [SVE Integer Reduction](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	0	opc	M	0	0	1	Pg				Zn				Zd						

Decode fields opc	Instruction Details
00	<a href="#">MOVPRFX (predicated)</a>
01	UNALLOCATED
1x	UNALLOCATED

**SVE bitwise logical reduction (predicated)**

These instructions are under [SVE Integer Reduction](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	0	opc	0	0	1	Pg				Zn				Vd						

Decode fields opc	Instruction Details
00	<a href="#">ORV</a>
01	<a href="#">EORV</a>
10	<a href="#">ANDV</a>
11	UNALLOCATED

**SVE bitwise logical reduction (quadwords)**

These instructions are under [SVE Integer Reduction](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	1	opc	0	0	1	Pg				Zn				Vd						

Decode fields opc	Instruction Details	Feature
00	<a href="#">ORQV</a>	FEAT_SVE2p1
01	<a href="#">EORQV</a>	FEAT_SVE2p1
10	<a href="#">ANDQV</a>	FEAT_SVE2p1
11	UNALLOCATED	-

**SVE Bitwise Shift - Predicated**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000100										0	op0					100															

### Decode fields op0

### Instruction details

0x	<a href="#">SVE bitwise shift by immediate (predicated)</a>
10	<a href="#">SVE bitwise shift by vector (predicated)</a>
11	<a href="#">SVE bitwise shift by wide elements (predicated)</a>

## SVE bitwise shift by immediate (predicated)

These instructions are under [SVE Bitwise Shift - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	tszh	0	0	opc	L	U	1	0	0	Pg		tszl		imm3								Zdn		

### Decode fields opc L U

### Instruction Details

00	0	0	<a href="#">ASR (immediate, predicated)</a>
00	0	1	<a href="#">LSR (immediate, predicated)</a>
00	1	0	UNALLOCATED
00	1	1	<a href="#">LSL (immediate, predicated)</a>
01	0	0	<a href="#">ASRD</a>
01	0	1	UNALLOCATED
01	1	0	<a href="#">SQSHL (immediate)</a>
01	1	1	<a href="#">UQSHL (immediate)</a>
10			UNALLOCATED
11	0	0	<a href="#">SRSHR</a>
11	0	1	<a href="#">URSHR</a>
11	1	0	UNALLOCATED
11	1	1	<a href="#">SQSHLU</a>

## SVE bitwise shift by vector (predicated)

These instructions are under [SVE Bitwise Shift - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	0	R	L	U	1	0	0	Pg	Zm			Zdn									

### Decode fields R L U

### Instruction Details

	1	0	UNALLOCATED
0	0	0	<a href="#">ASR (vectors)</a>
0	0	1	<a href="#">LSR (vectors)</a>
0	1	1	<a href="#">LSL (vectors)</a>
1	0	0	<a href="#">ASRR</a>
1	0	1	<a href="#">LSRR</a>
1	1	1	<a href="#">LSLR</a>

## SVE bitwise shift by wide elements (predicated)

These instructions are under [SVE Bitwise Shift - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	0	1	1	R	L	U	1	0	0	Pg	Zm			Zdn									

Decode fields			Instruction Details
R	L	U	
0	0	0	<a href="#">ASR (wide elements, predicated)</a>
0	0	1	<a href="#">LSR (wide elements, predicated)</a>
0	1	0	UNALLOCATED
0	1	1	<a href="#">LSL (wide elements, predicated)</a>
1			UNALLOCATED

## SVE Integer Unary Arithmetic - Predicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000100										0	op0						101														

Decode fields		Instruction details
op0		
0x		UNALLOCATED
10		<a href="#">SVE integer unary operations (predicated)</a>
11		<a href="#">SVE bitwise unary operations (predicated)</a>

## SVE integer unary operations (predicated)

These instructions are under [SVE Integer Unary Arithmetic - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size		0	1	0	opc			1	0	1		Pg				Zn					Zd		

Decode fields		Instruction Details
opc		
000		<a href="#">SXTB, SXTH, SXTW</a> — <a href="#">SXTB</a>
001		<a href="#">UXTB, UXTH, UXTW</a> — <a href="#">UXTB</a>
010		<a href="#">SXTB, SXTH, SXTW</a> — <a href="#">SXTH</a>
011		<a href="#">UXTB, UXTH, UXTW</a> — <a href="#">UXTH</a>
100		<a href="#">SXTB, SXTH, SXTW</a> — <a href="#">SXTW</a>
101		<a href="#">UXTB, UXTH, UXTW</a> — <a href="#">UXTW</a>
110		<a href="#">ABS</a>
111		<a href="#">NEG</a>

## SVE bitwise unary operations (predicated)

These instructions are under [SVE Integer Unary Arithmetic - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size		0	1	1	opc			1	0	1		Pg				Zn					Zd		

Decode fields	Instruction Details
opc	
000	<a href="#">CLS</a>
001	<a href="#">CLZ</a>
010	<a href="#">CNT</a>
011	<a href="#">CNOT</a>
100	<a href="#">FABS</a>
101	<a href="#">FNEG</a>

Decode fields	Instruction Details
opc	
110	<a href="#">NOT (vector)</a>
111	UNALLOCATED

### SVE integer add/subtract vectors (unpredicated)

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1			Zm				0	0	0	opc				Zn						Zd		

Decode fields	Instruction Details
opc	
000	<a href="#">ADD (vectors, unpredicated)</a>
001	<a href="#">SUB (vectors, unpredicated)</a>
01x	UNALLOCATED
100	<a href="#">SQADD (vectors, unpredicated)</a>
101	<a href="#">UQADD (vectors, unpredicated)</a>
110	<a href="#">SQSUB (vectors, unpredicated)</a>
111	<a href="#">UQSUB (vectors, unpredicated)</a>

### SVE Bitwise Logical - Unpredicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					00000100					1							001		op0												

Decode fields	Instruction details
op0	
0xx	UNALLOCATED
100	<a href="#">SVE bitwise logical operations (unpredicated)</a>
101	<a href="#">XAR</a>
11x	<a href="#">SVE2 bitwise ternary operations</a>

### SVE bitwise logical operations (unpredicated)

These instructions are under [SVE Bitwise Logical - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	opc	1			Zm				0	0	1	1	0	0			Zn					Zd		

Decode fields	Instruction Details
opc	
00	<a href="#">AND (vectors, unpredicated)</a>
01	<a href="#">ORR (vectors, unpredicated)</a>
10	<a href="#">EOR (vectors, unpredicated)</a>
11	<a href="#">BIC (vectors, unpredicated)</a>

### SVE2 bitwise ternary operations

These instructions are under [SVE Bitwise Logical - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	1	0	0	opc		1	Zm					0	0	1	1	1	o2		Zk					Zdn				

Decode fields		Instruction Details
opc	o2	
00	0	<a href="#">EOR3</a>
00	1	<a href="#">BSL</a>
01	0	<a href="#">BCAX</a>
01	1	<a href="#">BSL1N</a>
1x	0	UNALLOCATED
10	1	<a href="#">BSL2N</a>
11	1	<a href="#">NBSL</a>

## SVE Index Generation

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000100										1						0100		op0													

Decode fields		Instruction details
op0		
00		<a href="#">INDEX (immediates)</a>
01		<a href="#">INDEX (scalar, immediate)</a>
10		<a href="#">INDEX (immediate, scalar)</a>
11		<a href="#">INDEX (scalars)</a>

## SVE Stack Allocation

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000100								op0			1						0101		op1												

Decode fields		Instruction details
op0	op1	
0	0	<a href="#">SVE stack frame adjustment</a>
0	1	<a href="#">Streaming SVE stack frame adjustment</a>
1	0	<a href="#">SVE stack frame size</a>
1	1	<a href="#">Streaming SVE stack frame size</a>

## SVE stack frame adjustment

These instructions are under [SVE Stack Allocation](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	op		1	Rn					0	1	0	1	0	imm6					Rd				

Decode fields		Instruction Details
op		
0		<a href="#">ADDVL</a>
1		<a href="#">ADDPL</a>

**Streaming SVE stack frame adjustment**

These instructions are under [SVE Stack Allocation](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	op	1			Rn			0	1	0	1	1					imm6					Rd	

Decode fields op	Instruction Details	Feature
0	<a href="#">ADDSVL</a>	FEAT_SME
1	<a href="#">ADDSPL</a>	FEAT_SME

**SVE stack frame size**

These instructions are under [SVE Stack Allocation](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	op	1			opc2			0	1	0	1	0					imm6					Rd	

Decode fields op	Decode fields opc2	Instruction Details
0	0xxxxx	UNALLOCATED
0	10xxx	UNALLOCATED
0	110xx	UNALLOCATED
0	1110x	UNALLOCATED
0	11110	UNALLOCATED
0	11111	<a href="#">RDVL</a>
1		UNALLOCATED

**Streaming SVE stack frame size**

These instructions are under [SVE Stack Allocation](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	op	1			opc2			0	1	0	1	1					imm6					Rd	

Decode fields op	Decode fields opc2	Instruction Details	Feature
0	0xxxxx	UNALLOCATED	-
0	10xxx	UNALLOCATED	-
0	110xx	UNALLOCATED	-
0	1110x	UNALLOCATED	-
0	11110	UNALLOCATED	-
0	11111	<a href="#">RDSVL</a>	FEAT_SME
1		UNALLOCATED	-

**SVE2 Integer Multiply - Unpredicated**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										1											011	op0									

Decode fields op0	Instruction details
0x	<a href="#">SVE2 integer multiply vectors (unpredicated)</a>



10	<a href="#">SVE2 signed saturating doubling multiply high (unpredicated)</a>
11	UNALLOCATED

**SVE2 integer multiply vectors (unpredicated)**

These instructions are under [SVE2 Integer Multiply - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1				Zm			0	1	1	0	opc				Zn					Zd		

Decode fields		Instruction Details
size	opc	
	00	<a href="#">MUL (vectors, unpredicated)</a>
	10	<a href="#">SMULH (unpredicated)</a>
	11	<a href="#">UMULH (unpredicated)</a>
00	01	<a href="#">PMUL</a>
01	01	UNALLOCATED
1x	01	UNALLOCATED

**SVE2 signed saturating doubling multiply high (unpredicated)**

These instructions are under [SVE2 Integer Multiply - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1	Zm						0	1	1	1	0	R	Zn						Zd			

Decode fields		Instruction Details
R		
0		<a href="#">SQDMULH (vectors)</a>
1		<a href="#">SQRDMULH (vectors)</a>

**SVE Bitwise Shift - Unpredicated**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					00000100					1						100			op0												

Decode fields		Instruction details
op0		
0		<a href="#">SVE bitwise shift by wide elements (unpredicated)</a>
1		<a href="#">SVE bitwise shift by immediate (unpredicated)</a>

**SVE bitwise shift by wide elements (unpredicated)**

These instructions are under [SVE Bitwise Shift - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1				Zm			1	0	0	0	opc				Zn					Zd		

Decode fields		Instruction Details
opc		
00		<a href="#">ASR (wide elements, unpredicated)</a>
01		<a href="#">LSR (wide elements, unpredicated)</a>
10		UNALLOCATED

**Decode fields****opc****Instruction Details**

11	<a href="#">LSL (wide elements, unpredicated)</a>
----	---

**SVE bitwise shift by immediate (unpredicated)**

These instructions are under [SVE Bitwise Shift - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	tszh	1	tszl	imm3	1	0	0	1	opc	Zn						Zd								

**Decode fields****opc****Instruction Details**

00	<a href="#">ASR (immediate, unpredicated)</a>
01	<a href="#">LSR (immediate, unpredicated)</a>
10	UNALLOCATED
11	<a href="#">LSL (immediate, unpredicated)</a>

**SVE address generation**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	opc	1	Zm				1	0	1	0	msz	Zn				Zd								

**Decode fields****opc****Instruction Details**

00	<a href="#">ADR — Unpacked 32-bit signed offsets</a>
01	<a href="#">ADR — Unpacked 32-bit unsigned offsets</a>
1x	<a href="#">ADR — Packed offsets</a>

**SVE Integer Misc - Unpredicated**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
00000100										1					1011				op0																

**Decode fields****op0****Instruction details**

0x	<a href="#">SVE floating-point trig select coefficient</a>
10	<a href="#">SVE floating-point exponential accelerator</a>
11	<a href="#">SVE constructive prefix (unpredicated)</a>

**SVE floating-point trig select coefficient**

These instructions are under [SVE Integer Misc - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1	Zm				1	0	1	1	0	op	Zn				Zd							

**Decode fields****op****Instruction Details**

0	<a href="#">FTSSEL</a>
1	UNALLOCATED

**SVE floating-point exponential accelerator**

These instructions are under [SVE Integer Misc - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1				opc			1	0	1	1	1	0			Zn					Zd		

Decode fields opc	Instruction Details
00000	<a href="#">FEXPA</a>
00001	UNALLOCATED
0001x	UNALLOCATED
001xx	UNALLOCATED
01xxx	UNALLOCATED
1xxxx	UNALLOCATED

**SVE constructive prefix (unpredicated)**

These instructions are under [SVE Integer Misc - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	opc	1			opc2				1	0	1	1	1	1			Zn					Zd		

Decode fields opc	Decode fields opc2	Instruction Details
00	00000	<a href="#">MOVPRFX (unpredicated)</a>
00	00001	UNALLOCATED
00	0001x	UNALLOCATED
00	001xx	UNALLOCATED
00	01xxx	UNALLOCATED
00	1xxxx	UNALLOCATED
01		UNALLOCATED
1x		UNALLOCATED

**SVE Element Count**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
											1	op0					11			op1											

Decode fields op0	Decode fields op1	Instruction details
0	00x	<a href="#">SVE saturating inc/dec vector by element count</a>
0	100	<a href="#">SVE element count</a>
0	101	UNALLOCATED
1	000	<a href="#">SVE inc/dec vector by element count</a>
1	100	<a href="#">SVE inc/dec register by element count</a>
1	x01	UNALLOCATED
	01x	UNALLOCATED
	11x	<a href="#">SVE saturating inc/dec register by element count</a>

**SVE saturating inc/dec vector by element count**

These instructions are under [SVE Element Count](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1	0	imm4				1	1	0	0	D	U	pattern				Zdn						

**Decode fields**  
size      D      U      **Instruction Details**

00			UNALLOCATED
01	0	0	<a href="#">SQINCH (vector)</a>
01	0	1	<a href="#">UQINCH (vector)</a>
01	1	0	<a href="#">SQDECH (vector)</a>
01	1	1	<a href="#">UQDECH (vector)</a>
10	0	0	<a href="#">SQINCW (vector)</a>
10	0	1	<a href="#">UQINCW (vector)</a>
10	1	0	<a href="#">SQDECW (vector)</a>
10	1	1	<a href="#">UQDECW (vector)</a>
11	0	0	<a href="#">SQINCD (vector)</a>
11	0	1	<a href="#">UQINCD (vector)</a>
11	1	0	<a href="#">SQDECD (vector)</a>
11	1	1	<a href="#">UQDECD (vector)</a>

### SVE element count

These instructions are under [SVE Element Count](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size		1	0	imm4				1	1	1	0	0	op	pattern					Rd				

**Decode fields**  
size      op      **Instruction Details**

	1	UNALLOCATED
00	0	<a href="#">CNTB, CNTD, CNTH, CNTW</a> — <a href="#">CNTB</a>
01	0	<a href="#">CNTB, CNTD, CNTH, CNTW</a> — <a href="#">CNTH</a>
10	0	<a href="#">CNTB, CNTD, CNTH, CNTW</a> — <a href="#">CNTW</a>
11	0	<a href="#">CNTB, CNTD, CNTH, CNTW</a> — <a href="#">CNTD</a>

### SVE inc/dec vector by element count

These instructions are under [SVE Element Count](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1	1	imm4				1	1	0	0	0	D	pattern					Zdn					

**Decode fields**  
size      D      **Instruction Details**

00		UNALLOCATED
01	0	<a href="#">INCD, INCH, INCW (vector)</a> — <a href="#">INCH</a>
01	1	<a href="#">DECD, DECH, DECW (vector)</a> — <a href="#">DECH</a>
10	0	<a href="#">INCD, INCH, INCW (vector)</a> — <a href="#">INCW</a>
10	1	<a href="#">DECD, DECH, DECW (vector)</a> — <a href="#">DECW</a>
11	0	<a href="#">INCD, INCH, INCW (vector)</a> — <a href="#">INCD</a>
11	1	<a href="#">DECD, DECH, DECW (vector)</a> — <a href="#">DECD</a>

### SVE inc/dec register by element count

These instructions are under [SVE Element Count](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1	1		imm4	1	1	1	0	0	D		pattern											Rdn

Decode fields		Instruction Details
size	D	
00	0	<a href="#">INCB, INCD, INCH, INCW (scalar)</a> — <a href="#">INCB</a>
00	1	<a href="#">DECB, DECD, DECH, DECW (scalar)</a> — <a href="#">DECB</a>
01	0	<a href="#">INCB, INCD, INCH, INCW (scalar)</a> — <a href="#">INCH</a>
01	1	<a href="#">DECB, DECD, DECH, DECW (scalar)</a> — <a href="#">DECH</a>
10	0	<a href="#">INCB, INCD, INCH, INCW (scalar)</a> — <a href="#">INCW</a>
10	1	<a href="#">DECB, DECD, DECH, DECW (scalar)</a> — <a href="#">DECW</a>
11	0	<a href="#">INCB, INCD, INCH, INCW (scalar)</a> — <a href="#">INCD</a>
11	1	<a href="#">DECB, DECD, DECH, DECW (scalar)</a> — <a href="#">DECD</a>

**SVE saturating inc/dec register by element count**

These instructions are under [SVE Element Count](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size	1	sf		imm4	1	1	1	1	D	U		pattern											Rdn

Decode fields				Instruction Details
size	sf	D	U	
00	0	0	0	<a href="#">SQINCB</a> — 32-bit
00	0	0	1	<a href="#">UQINCB</a> — 32-bit
00	0	1	0	<a href="#">SQDECB</a> — 32-bit
00	0	1	1	<a href="#">UQDECB</a> — 32-bit
00	1	0	0	<a href="#">SQINCB</a> — 64-bit
00	1	0	1	<a href="#">UQINCB</a> — 64-bit
00	1	1	0	<a href="#">SQDECB</a> — 64-bit
00	1	1	1	<a href="#">UQDECB</a> — 64-bit
01	0	0	0	<a href="#">SQINCH (scalar)</a> — 32-bit
01	0	0	1	<a href="#">UQINCH (scalar)</a> — 32-bit
01	0	1	0	<a href="#">SQDECH (scalar)</a> — 32-bit
01	0	1	1	<a href="#">UQDECH (scalar)</a> — 32-bit
01	1	0	0	<a href="#">SQINCH (scalar)</a> — 64-bit
01	1	0	1	<a href="#">UQINCH (scalar)</a> — 64-bit
01	1	1	0	<a href="#">SQDECH (scalar)</a> — 64-bit
01	1	1	1	<a href="#">UQDECH (scalar)</a> — 64-bit
10	0	0	0	<a href="#">SQINCW (scalar)</a> — 32-bit
10	0	0	1	<a href="#">UQINCW (scalar)</a> — 32-bit
10	0	1	0	<a href="#">SQDECW (scalar)</a> — 32-bit
10	0	1	1	<a href="#">UQDECW (scalar)</a> — 32-bit
10	1	0	0	<a href="#">SQINCW (scalar)</a> — 64-bit
10	1	0	1	<a href="#">UQINCW (scalar)</a> — 64-bit
10	1	1	0	<a href="#">SQDECW (scalar)</a> — 64-bit
10	1	1	1	<a href="#">UQDECW (scalar)</a> — 64-bit
11	0	0	0	<a href="#">SQINCD (scalar)</a> — 32-bit
11	0	0	1	<a href="#">UQINCD (scalar)</a> — 32-bit
11	0	1	0	<a href="#">SQDECD (scalar)</a> — 32-bit
11	0	1	1	<a href="#">UQDECD (scalar)</a> — 32-bit
11	1	0	0	<a href="#">SQINCD (scalar)</a> — 64-bit

Decode fields				Instruction Details
size	sf	D	U	
11	1	0	1	<a href="#">UQINCD (scalar) — 64-bit</a>
11	1	1	0	<a href="#">SQDECD (scalar) — 64-bit</a>
11	1	1	1	<a href="#">UQDECD (scalar) — 64-bit</a>

## SVE Bitwise Immediate

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000101								op0		00		op1																			

Decode fields		Instruction details
op0	op1	
11	00	<a href="#">DUPM</a>
!= 11	00	<a href="#">SVE bitwise logical with immediate (unpredicated)</a>
	!= 00	UNALLOCATED

## SVE bitwise logical with immediate (unpredicated)

These instructions are under [SVE Bitwise Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	!= 11		0	0	0	0	imm13										Zdn							
								opc																							

The following constraints also apply to this encoding: `opc != 11 && opc != 11`

Decode fields	Instruction Details
opc	
00	<a href="#">ORR (immediate)</a>
01	<a href="#">EOR (immediate)</a>
10	<a href="#">AND (immediate)</a>

## SVE Integer Wide Immediate - Predicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000101										01						op0															

Decode fields	Instruction details
op0	
0xx	<a href="#">SVE copy integer immediate (predicated)</a>
10x	UNALLOCATED
110	<a href="#">FCPY</a>
111	UNALLOCATED

## SVE copy integer immediate (predicated)

These instructions are under [SVE Integer Wide Immediate - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size		0	1	Pg		0	M	sh	imm8						Zd								

Decode fields M	Instruction Details
0	<a href="#">CPY (immediate, zeroing)</a>
1	<a href="#">CPY (immediate, merging)</a>

## SVE Permute Vector - One Source Quadwords

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000101								op0		1	op1					001001															

Decode fields op0	Decode fields op1	Instruction details	Feature
00		<a href="#">DUPO</a>	FEAT_SVE2p1
01	0	<a href="#">EXTQ</a>	FEAT_SVE2p1
01	1	UNALLOCATED	-
1x		UNALLOCATED	-

## SVE table lookup (three sources)

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size		1	Zm				0	0	1	0	1	op	Zn				Zd						

Decode fields op	Instruction Details
0	<a href="#">TBL</a>
1	<a href="#">TBX</a>

## SVE Permute Vector - Unpredicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000101									1	op0	op1	001110				op2			op3												

op0	Decode fields op1	Decode fields op2	Decode fields op3	Instruction details
00	000			<a href="#">DUP (scalar)</a>
00	100			<a href="#">INSR (scalar)</a>
00	x10			UNALLOCATED
00	xx1			UNALLOCATED
01	xx0		0	<a href="#">SVE move predicate from vector</a>
01	xx0		1	UNALLOCATED
01	xx1	0		<a href="#">SVE move predicate into vector</a>
01	xx1	1		UNALLOCATED
10	0xx			<a href="#">SVE unpack vector elements</a>
10	100			<a href="#">INSR (SIMD&amp;FP scalar)</a>
10	110			UNALLOCATED
10	1x1			UNALLOCATED
11	000			<a href="#">REV (vector)</a>

11	!= 000			UNALLOCATED
----	--------	--	--	-------------

## SVE move predicate from vector

These instructions are under [SVE Permute Vector - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	opc	1	0	1	opc2	0	0	0	1	1	1	0	Zn					0	Pd					

Decode fields		Instruction Details	Feature
opc	opc2		
00	00	UNALLOCATED	-
00	01	<a href="#">PMOV (to predicate)</a> — <a href="#">byte</a>	FEAT_SVE2p1
00	1x	<a href="#">PMOV (to predicate)</a> — <a href="#">halfword</a>	FEAT_SVE2p1
01		<a href="#">PMOV (to predicate)</a> — <a href="#">word</a>	FEAT_SVE2p1
1x		<a href="#">PMOV (to predicate)</a> — <a href="#">doubleword</a>	FEAT_SVE2p1

## SVE move predicate into vector

These instructions are under [SVE Permute Vector - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	opc		1	0	1	opc2		1	0	0	1	1	1	0	0	Pn				Zd				

Decode opc	fields opc2	Instruction Details	Feature
00	00	UNALLOCATED	-
00	01	<a href="#">PMOV (to vector) — byte</a>	FEAT_SVE2p1
00	1x	<a href="#">PMOV (to vector) — halfword</a>	FEAT_SVE2p1
01		<a href="#">PMOV (to vector) — word</a>	FEAT_SVE2p1
1x		<a href="#">PMOV (to vector) — doubleword</a>	FEAT_SVE2p1

## SVE unpack vector elements

These instructions are under [SVE Permute Vector - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	1	0	0	U	H	0	0	1	1	1	0	Zn					Zd					

Decode fields		Instruction Details
U	H	
0	0	<a href="#">SUNPKHL, SUNPKLO</a> — <a href="#">SUNPKLO</a>
0	1	<a href="#">SUNPKHL, SUNPKLO</a> — <a href="#">SUNPKHI</a>
1	0	<a href="#">UUNPKHL, UUNPKLO</a> — <a href="#">UUNPKLO</a>
1	1	<a href="#">UUNPKHL, UUNPKLO</a> — <a href="#">UUNPKHI</a>

## SVE Permute Predicate

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000101								op0		1	op1				010		op2								op3						

op0	Decode fields			Instruction details
	op1	op2	op3	
00000000	00000000	00000000	00000000	ADDI rd, rs1, imm
00000001	00000000	00000000	00000000	SLTI rd, rs1, imm
00000010	00000000	00000000	00000000	SLTIU rd, rs1, imm
00000011	00000000	00000000	00000000	ANDI rd, rs1, imm
00000100	00000000	00000000	00000000	ORR rd, rs1, rs2
00000101	00000000	00000000	00000000	AND rd, rs1, rs2
00000110	00000000	00000000	00000000	XOR rd, rs1, rs2
00000111	00000000	00000000	00000000	SLL rd, rs1, shamt
00001000	00000000	00000000	00000000	SRL rd, rs1, shamt
00001001	00000000	00000000	00000000	SHL rd, rs1, shamt
00001010	00000000	00000000	00000000	SHR rd, rs1, shamt
00001011	00000000	00000000	00000000	SHRUI rd, rs1, shamt
00001100	00000000	00000000	00000000	SHLL rd, rs1, shamt
00001101	00000000	00000000	00000000	SHRL rd, rs1, shamt
00001110	00000000	00000000	00000000	SHRUI rd, rs1, shamt
00001111	00000000	00000000	00000000	SHLL rd, rs1, shamt
00010000	00000000	00000000	00000000	SHLL rd, rs1, shamt
00010001	00000000	00000000	00000000	SHRL rd, rs1, shamt
00010010	00000000	00000000	00000000	SHRUI rd, rs1, shamt
00010011	00000000	00000000	00000000	SHLL rd, rs1, shamt
00010100	00000000	00000000	00000000	SHRL rd, rs1, shamt
00010101	00000000	00000000	00000000	SHRUI rd, rs1, shamt
00010110	00000000	00000000	00000000	SHLL rd, rs1, shamt
00010111	00000000	00000000	00000000	SHRL rd, rs1, shamt
00011000	00000000	00000000	00000000	SHRUI rd, rs1, shamt
00011001	00000000	00000000	00000000	SHLL rd, rs1, shamt
00011010	00000000	00000000	00000000	SHRL rd, rs1, shamt
00011011	00000000	00000000	00000000	SHRUI rd, rs1, shamt
00011100	00000000	00000000	00000000	SHLL rd, rs1, shamt
00011101	00000000	00000000	00000000	SHRL rd, rs1, shamt
00011110	00000000	00000000	00000000	SHRUI rd, rs1, shamt
00011111	00000000	00000000	00000000	SHLL rd, rs1, shamt
00100000	00000000	00000000	00000000	SHRL rd, rs1, shamt
00100001	00000000	00000000	00000000	SHRUI rd, rs1, shamt
00100010	00000000	00000000	00000000	SHLL rd, rs1, shamt
00100011	00000000	00000000	00000000	SHRL rd, rs1, shamt
00100100	00000000	00000000	00000000	SHRUI rd, rs1, shamt
00100101	00000000	00000000	00000000	SHLL rd, rs1, shamt
00100110	00000000	00000000	00000000	SHRL rd, rs1, shamt
00100111	00000000	00000000	00000000	SHRUI rd, rs1, shamt
00101000	00000000	00000000	00000000	SHLL rd, rs1, shamt
00101001	00000000	00000000	00000000	SHRL rd, rs1, shamt
00101010	00000000	00000000	00000000	SHRUI rd, rs1, shamt
00101011	00000000	00000000	00000000	SHLL rd, rs1, shamt
00101100	00000000	00000000	00000000	SHRL rd, rs1, shamt
00101101	00000000	00000000	00000000	SHRUI rd, rs1, shamt
00101110	00000000	00000000	00000000	SHLL rd, rs1, shamt
00101111	00000000	00000000	00000000	SHRL rd, rs1, shamt
00110000	00000000	00000000	00000000	SHRUI rd, rs1, shamt
00110001	00000000	00000000	00000000	SHLL rd, rs1, shamt
00110010	00000000	00000000	00000000	SHRL rd, rs1, shamt
00110011	00000000	00000000	00000000	SHRUI rd, rs1, shamt
00110100	00000000	00000000	00000000	SHLL rd, rs1, shamt
00110101	00000000	00000000	00000000	SHRL rd, rs1, shamt
00110110	00000000	00000000	00000000	SHRUI rd, rs1, shamt
00110111	00000000	00000000	00000000	SHLL rd, rs1, shamt
00111000	00000000	00000000	00000000	SHRL rd, rs1, shamt
00111001	00000000	00000000	00000000	SHRUI rd, rs1, shamt
00111010	00000000	00000000	00000000	SHLL rd, rs1, shamt
00111011	00000000	00000000	00000000	SHRL rd, rs1, shamt
00111100	00000000	00000000	00000000	SHRUI rd, rs1, shamt
00111101	00000000	00000000	00000000	SHLL rd, rs1, shamt
00111110	00000000	00000000	00000000	SHRL rd, rs1, shamt
00111111	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01000000	00000000	00000000	00000000	SHLL rd, rs1, shamt
01000001	00000000	00000000	00000000	SHRL rd, rs1, shamt
01000010	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01000011	00000000	00000000	00000000	SHLL rd, rs1, shamt
01000100	00000000	00000000	00000000	SHRL rd, rs1, shamt
01000101	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01000110	00000000	00000000	00000000	SHLL rd, rs1, shamt
01000111	00000000	00000000	00000000	SHRL rd, rs1, shamt
01001000	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01001001	00000000	00000000	00000000	SHLL rd, rs1, shamt
01001010	00000000	00000000	00000000	SHRL rd, rs1, shamt
01001011	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01001100	00000000	00000000	00000000	SHLL rd, rs1, shamt
01001101	00000000	00000000	00000000	SHRL rd, rs1, shamt
01001110	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01001111	00000000	00000000	00000000	SHLL rd, rs1, shamt
01010000	00000000	00000000	00000000	SHRL rd, rs1, shamt
01010001	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01010010	00000000	00000000	00000000	SHLL rd, rs1, shamt
01010011	00000000	00000000	00000000	SHRL rd, rs1, shamt
01010100	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01010101	00000000	00000000	00000000	SHLL rd, rs1, shamt
01010110	00000000	00000000	00000000	SHRL rd, rs1, shamt
01010111	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01011000	00000000	00000000	00000000	SHLL rd, rs1, shamt
01011001	00000000	00000000	00000000	SHRL rd, rs1, shamt
01011010	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01011011	00000000	00000000	00000000	SHLL rd, rs1, shamt
01011100	00000000	00000000	00000000	SHRL rd, rs1, shamt
01011101	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01011110	00000000	00000000	00000000	SHLL rd, rs1, shamt
01011111	00000000	00000000	00000000	SHRL rd, rs1, shamt
01100000	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01100001	00000000	00000000	00000000	SHLL rd, rs1, shamt
01100010	00000000	00000000	00000000	SHRL rd, rs1, shamt
01100011	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01100100	00000000	00000000	00000000	SHLL rd, rs1, shamt
01100101	00000000	00000000	00000000	SHRL rd, rs1, shamt
01100110	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01100111	00000000	00000000	00000000	SHLL rd, rs1, shamt
01101000	00000000	00000000	00000000	SHRL rd, rs1, shamt
01101001	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01101010	00000000	00000000	00000000	SHLL rd, rs1, shamt
01101011	00000000	00000000	00000000	SHRL rd, rs1, shamt
01101100	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01101101	00000000	00000000	00000000	SHLL rd, rs1, shamt
01101110	00000000	00000000	00000000	SHRL rd, rs1, shamt
01101111	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01110000	00000000	00000000	00000000	SHLL rd, rs1, shamt
01110001	00000000	00000000	00000000	SHRL rd, rs1, shamt
01110010	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01110011	00000000	00000000	00000000	SHLL rd, rs1, shamt
01110100	00000000	00000000	00000000	SHRL rd, rs1, shamt
01110101	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01110110	00000000	00000000	00000000	SHLL rd, rs1, shamt
01110111	00000000	00000000	00000000	SHRL rd, rs1, shamt
01111000	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01111001	00000000	00000000	00000000	SHLL rd, rs1, shamt
01111010	00000000	00000000	00000000	SHRL rd, rs1, shamt
01111011	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01111100	00000000	00000000	00000000	SHLL rd, rs1, shamt
01111101	00000000	00000000	00000000	SHRL rd, rs1, shamt
01111110	00000000	00000000	00000000	SHRUI rd, rs1, shamt
01111111	00000000	00000000	00000000	SHLL rd, rs1, shamt
10000000	00000000	00000000	00000000	SHRL rd, rs1, shamt
10000001	00000000	00000000	00000000	SHRUI rd, rs1, shamt
10000010	00000000	00000000	00000000	SHLL rd, rs1, shamt
10000011	00000000	00000000	00000000	SHRL rd, rs1, shamt
10000100	00000000	00000000	00000000	SHRUI rd, rs1, shamt
10000101	00000000	00000000	00000000	SHLL rd, rs1, shamt
10000110	00000000	00000000	00000000	SHRL rd, rs1, shamt
10000111	00000000	00000000	00000000	SHRUI rd, rs1, shamt
10001000	00000000	00000000	00000000	SHLL rd, rs1, shamt
10001001	00000000	00000000	00000000	SHRL rd, rs1, shamt
10001010	00000000	00000000	00000000	SHRUI rd, rs1, shamt
10001011	00000000	00000000	00000000	SHLL rd, rs1, shamt
10001100	00000000	00000000	00000000	SHRL rd, rs1, shamt
10001101	00000000	00000000	00000000	SHRUI rd, rs1, shamt
10001110	00000000	00000000	00000000	SHLL rd, rs1, shamt
10001111	00000000	00000000	00000000	SHRL rd, rs1, shamt
10010000	00000000	00000000	00000000	SHRUI rd, rs1, shamt
10010001	00000000	00000000	00000000	SHLL rd, rs1, shamt
10010010	00000000	00000000	00000000	SHRL rd, rs1, shamt
10010011	00000000	00000000	00000000	SHRUI rd, rs1, shamt
10010100	00000000	00000000	00000000	SHLL rd, rs1, shamt
10010101	00000000	00000000	00000000	SHRL rd, rs1, shamt
10010110	00000000	00000000	00000000	SHRUI rd, rs1, shamt
10010111	00000000	00000000	00000000	SHLL rd, rs1, shamt
10011000	00000000	00000000	00000000	SHRL rd, rs1, shamt
10011001	00000000	00000000	00000000	SHRUI rd, rs1, shamt
10011010	00000000	00000000	00000000	SHLL rd, rs1, shamt
10011011	00000000	00000000	00000000	SHRL rd, rs1, shamt
10011100	00000000	00000000	00000000	SHRUI rd, rs1, shamt
10011101	00000000	00000000	00000000	SHLL rd, rs1, shamt
10011110	00000000	00000000	00000000	SHRL rd, rs1, shamt
10011111	00000000	00000000	00000000	SHRUI rd, rs1, shamt
10100000	00000000	00000000	00000000	SHLL rd, rs1, shamt
10100001	00000000	00000000	00000000	SHRL rd, rs1, shamt
10100010	00000000	00000000	00000000	SHRUI rd, rs1, shamt
10100011	00000000	00000000	00000000	SHLL rd, rs1, shamt
10100100	00000000	00000000	00000000	SHRL rd, rs1, shamt
10100101	00000000	00000000	00000000	SHRUI rd, rs1, shamt
10100110	00000000	00000000	00000000	SHLL rd, rs1, shamt
10100111	00000000	00000000	00000000	SHRL rd, rs1, shamt
10101000	00000000	00000000	00000000	SHRUI rd, rs1, shamt
10101001	00000000	00000000	00000000	SHLL rd, rs1, shamt
10101010	00000000	00000000	00000000	SHRL rd, rs1, shamt
10101011	00000000	00000000	00000000	SHRUI rd, rs1, shamt
10101100	00000000	00000000	00000000	SHLL rd, rs1, shamt
10101101	00000000	00000000	00000000	SHRL rd, rs1, shamt
10101110	00000000	00000000	00000000	SHRUI rd, rs1, shamt
10101111	00000000			



00	1000x	0000	0	<a href="#">SVE unpack predicate elements</a>
01	1000x	0000	0	UNALLOCATED
10	1000x	0000	0	UNALLOCATED
11	1000x	0000	0	UNALLOCATED
	0xxxx	xxx0	0	<a href="#">SVE permute predicate elements</a>
	0xxxx	xxx1	0	UNALLOCATED
	10100	0000	0	<a href="#">REV (predicate)</a>
	10101	0000	0	UNALLOCATED
	10x0x	1000	0	UNALLOCATED
	10x0x	x100	0	UNALLOCATED
	10x0x	xx10	0	UNALLOCATED
	10x0x	xxx1	0	UNALLOCATED
	10x1x		0	UNALLOCATED
	11xxx		0	UNALLOCATED
			1	UNALLOCATED

**SVE unpack predicate elements**

These instructions are under [SVE Permute Predicate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	0	1	1	0	0	0	H	0	1	0	0	0	0	0		Pn		0			Pd		

**Decode fields****H****Instruction Details**

0	<a href="#">PUNPKHI, PUNPKLO — PUNPKLO</a>
1	<a href="#">PUNPKHL, PUNPKLO — PUNPKHI</a>

**SVE permute predicate elements**

These instructions are under [SVE Permute Predicate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size		1	0	Pm			0		1	0	opc		H	0	Pn			0		Pd			

**Decode fields****opc****H****Instruction Details**

00	0	<a href="#">ZIP1, ZIP2 (predicates) — ZIP1</a>
00	1	<a href="#">ZIP1, ZIP2 (predicates) — ZIP2</a>
01	0	<a href="#">UZP1, UZP2 (predicates) — UZP1</a>
01	1	<a href="#">UZP1, UZP2 (predicates) — UZP2</a>
10	0	<a href="#">TRN1, TRN2 (predicates) — TRN1</a>
10	1	<a href="#">TRN1, TRN2 (predicates) — TRN2</a>
11		UNALLOCATED

**SVE permute vector elements**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	Zm			0	1	1	opc		Zn				Zd									

Decode fields	Instruction Details
opc	
000	<a href="#">ZIP1, ZIP2 (vectors) — ZIP1</a>
001	<a href="#">ZIP1, ZIP2 (vectors) — ZIP2</a>
010	<a href="#">UZP1, UZP2 (vectors) — UZP1</a>
011	<a href="#">UZP1, UZP2 (vectors) — UZP2</a>
100	<a href="#">TRN1, TRN2 (vectors) — TRN1</a>
101	<a href="#">TRN1, TRN2 (vectors) — TRN2</a>
11x	UNALLOCATED

## SVE Permute Vector - Predicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	1	0	0	0	1	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Decode fields				Instruction details
op0	op1	op2	op3	
0	000	0	0	<a href="#">CPY (SIMD&amp;FP scalar)</a>
0	000	1	0	<a href="#">COMPACT</a>
0	000		1	<a href="#">SVE extract element to general register</a>
0	001		0	<a href="#">SVE extract element to SIMD&amp;FP scalar register</a>
0	01x		0	<a href="#">SVE reverse within elements</a>
0	01x		1	UNALLOCATED
0	100	0	1	<a href="#">CPY (scalar)</a>
0	100	1	1	UNALLOCATED
0	100		0	<a href="#">SVE conditionally broadcast element to vector</a>
0	101		0	<a href="#">SVE conditionally extract element to SIMD&amp;FP scalar</a>
0	110	0	0	<a href="#">SPLICE — Destructive</a>
0	110	1	0	<a href="#">SPLICE — Constructive</a>
0	110		1	UNALLOCATED
0	111	0	0	<a href="#">SVE reverse doublewords</a>
0	111	0	1	UNALLOCATED
0	111	1		UNALLOCATED
0	x01		1	UNALLOCATED
1	000		0	UNALLOCATED
1	000		1	<a href="#">SVE conditionally extract element to general register</a>
1	!= 000			UNALLOCATED

## SVE extract element to general register

These instructions are under [SVE Permute Vector - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	0	0	0	0	0	B	1	0	1	Pg				Zn						Rd		

Decode fields	Instruction Details
B	
0	<a href="#">LASTA (scalar)</a>
1	<a href="#">LASTB (scalar)</a>

**SVE extract element to SIMD&FP scalar register**

These instructions are under [SVE Permute Vector - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	0	0	0	1	B	1	0	0	Pg	Zn			Vd									

Decode fields B	Instruction Details
0	<a href="#">LASTA (SIMD&amp;FP scalar)</a>
1	<a href="#">LASTB (SIMD&amp;FP scalar)</a>

**SVE reverse within elements**

These instructions are under [SVE Permute Vector - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	0	0	1	opc	1	0	0	Pg				Zn				Zd						

Decode fields opc	Instruction Details
00	<a href="#">REVB, REVH, REVW — REVB</a>
01	<a href="#">REVB, REVH, REVW — REVH</a>
10	<a href="#">REVB, REVH, REVW — REVW</a>
11	<a href="#">RBIT</a>

**SVE conditionally broadcast element to vector**

These instructions are under [SVE Permute Vector - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	0	1	0	0	B	1	0	0	Pg	Zm				Zdn								

Decode fields B	Instruction Details
0	<a href="#">CLASTA (vectors)</a>
1	<a href="#">CLASTB (vectors)</a>

**SVE conditionally extract element to SIMD&FP scalar**

These instructions are under [SVE Permute Vector - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	0	1	0	1	B	1	0	0	Pg	Zm				Vdn								

Decode fields B	Instruction Details
0	<a href="#">CLASTA (SIMD&amp;FP scalar)</a>
1	<a href="#">CLASTB (SIMD&amp;FP scalar)</a>

**SVE reverse doublewords**

These instructions are under [SVE Permute Vector - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	0	1	1	1	0	1	0	0	Pg	Zn				Zd								

Decode fields size	Instruction Details	Feature
00	<a href="#">REVD</a>	FEAT_SVE2p1
01	UNALLOCATED	-
1x	UNALLOCATED	-

### SVE conditionally extract element to general register

These instructions are under [SVE Permute Vector - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size	1	1	0	0	0	B	1	0	1	Pg	Zm	Rdn											

Decode fields B	Instruction Details
0	<a href="#">CLASTA (scalar)</a>
1	<a href="#">CLASTB (scalar)</a>

### SVE Permute Vector - Extract

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	op0	1							0	0	0													

Decode fields op0	Instruction details
0	<a href="#">EXT</a> — <a href="#">Destructive</a>
1	<a href="#">EXT</a> — <a href="#">Constructive</a>

### SVE Permute Vector - Segments

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	op0	1							0	0	0													

Decode fields op0	Instruction details
0	<a href="#">SVE permute vector segments</a>
1	UNALLOCATED

### SVE permute vector segments

These instructions are under [SVE Permute Vector - Segments](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	1	0	1	Zm					0	0	0	opc	H	Zn					Zd					

Decode fields opc	Decode fields H	Instruction Details	Feature
00	0	<a href="#">ZIP1, ZIP2 (vectors)</a> — <a href="#">ZIP1</a>	FEAT_F64MM
00	1	<a href="#">ZIP1, ZIP2 (vectors)</a> — <a href="#">ZIP2</a>	FEAT_F64MM
01	0	<a href="#">UZP1, UZP2 (vectors)</a> — <a href="#">UZP1</a>	FEAT_F64MM
01	1	<a href="#">UZP1, UZP2 (vectors)</a> — <a href="#">UZP2</a>	FEAT_F64MM

Decode fields		Instruction Details	Feature
opc	H		
10		UNALLOCATED	-
11	0	<a href="#">TRN1, TRN2 (vectors)</a> — <a href="#">TRN1</a>	FEAT_F64MM
11	1	<a href="#">TRN1, TRN2 (vectors)</a> — <a href="#">TRN2</a>	FEAT_F64MM

## SVE Integer Compare - Vectors

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00100100										0							op0														

Decode fields		Instruction details
op0		
0		<a href="#">SVE integer compare vectors</a>
1		<a href="#">SVE integer compare with wide elements</a>

### SVE integer compare vectors

These instructions are under [SVE Integer Compare - Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size	0	Zm			op	0	o2	Pg		Zn				ne	Pd								

Decode fields			Instruction Details
op	o2	ne	
0	0	0	<a href="#">CMP&lt;cc&gt; (vectors)</a> — <a href="#">CMPHS</a>
0	0	1	<a href="#">CMP&lt;cc&gt; (vectors)</a> — <a href="#">CMPHI</a>
0	1	0	<a href="#">CMP&lt;cc&gt; (wide elements)</a> — <a href="#">CMPEQ</a>
0	1	1	<a href="#">CMP&lt;cc&gt; (wide elements)</a> — <a href="#">CMPNE</a>
1	0	0	<a href="#">CMP&lt;cc&gt; (vectors)</a> — <a href="#">CMPGE</a>
1	0	1	<a href="#">CMP&lt;cc&gt; (vectors)</a> — <a href="#">CMPGT</a>
1	1	0	<a href="#">CMP&lt;cc&gt; (vectors)</a> — <a href="#">CMPEQ</a>
1	1	1	<a href="#">CMP&lt;cc&gt; (vectors)</a> — <a href="#">CMPNE</a>

### SVE integer compare with wide elements

These instructions are under [SVE Integer Compare - Vectors](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size	0	Zm			U	1	lt	Pg		Zn			ne		Pd								

Decode fields			Instruction Details
U	lt	ne	
0	0	0	<a href="#">CMP&lt;cc&gt; (wide elements)</a> — <a href="#">CMPGE</a>
0	0	1	<a href="#">CMP&lt;cc&gt; (wide elements)</a> — <a href="#">CMPGT</a>
0	1	0	<a href="#">CMP&lt;cc&gt; (wide elements)</a> — <a href="#">CMPLT</a>
0	1	1	<a href="#">CMP&lt;cc&gt; (wide elements)</a> — <a href="#">CMPLE</a>
1	0	0	<a href="#">CMP&lt;cc&gt; (wide elements)</a> — <a href="#">CMPHS</a>
1	0	1	<a href="#">CMP&lt;cc&gt; (wide elements)</a> — <a href="#">CMPHI</a>
1	1	0	<a href="#">CMP&lt;cc&gt; (wide elements)</a> — <a href="#">CMPLO</a>
1	1	1	<a href="#">CMP&lt;cc&gt; (wide elements)</a> — <a href="#">CMPLS</a>

**SVE integer compare with unsigned immediate**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	size	1	imm7				lt	Pg		Zn				ne	Pd									

Decode fields		Instruction Details
lt	ne	
0	0	<a href="#">CMP&lt;cc&gt; (immediate)</a> — CMPHS
0	1	<a href="#">CMP&lt;cc&gt; (immediate)</a> — CMPHI
1	0	<a href="#">CMP&lt;cc&gt; (immediate)</a> — CMPLO
1	1	<a href="#">CMP&lt;cc&gt; (immediate)</a> — CMPLS

**SVE integer compare with signed immediate**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	0	imm5				op	0	o2	Pg		Zn				ne	Pd							

Decode fields			Instruction Details
op	o2	ne	
0	0	0	<a href="#">CMP&lt;cc&gt; (immediate)</a> — CMPGE
0	0	1	<a href="#">CMP&lt;cc&gt; (immediate)</a> — CMPGT
0	1	0	<a href="#">CMP&lt;cc&gt; (immediate)</a> — CMPLT
0	1	1	<a href="#">CMP&lt;cc&gt; (immediate)</a> — CMPLE
1	0	0	<a href="#">CMP&lt;cc&gt; (immediate)</a> — CMPEQ
1	0	1	<a href="#">CMP&lt;cc&gt; (immediate)</a> — CMPNE
1	1		UNALLOCATED

**SVE predicate logical operations**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	op	S	0	0	Pm				0	1	Pg		o2	Pn				o3	Pd					

Decode fields				Instruction Details
op	S	o2	o3	
0	0	0	0	<a href="#">AND (predicates)</a>
0	0	0	1	<a href="#">BIC (predicates)</a>
0	0	1	0	<a href="#">EOR (predicates)</a>
0	0	1	1	<a href="#">SEL (predicates)</a>
0	1	0	0	<a href="#">ANDS</a>
0	1	0	1	<a href="#">BICS</a>
0	1	1	0	<a href="#">EORS</a>
0	1	1	1	UNALLOCATED
1	0	0	0	<a href="#">ORR (predicates)</a>
1	0	0	1	<a href="#">ORN (predicates)</a>
1	0	1	0	<a href="#">NOR</a>
1	0	1	1	<a href="#">NAND</a>
1	1	0	0	<a href="#">ORRS</a>
1	1	0	1	<a href="#">ORNS</a>
1	1	1	0	<a href="#">NORS</a>

Decode fields				Instruction Details
op	S	o2	o3	
1	1	1	1	<a href="#">NANDS</a>

## SVE Propagate Break

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00100101										00						11						op0									

Decode fields		Instruction details
op0		
0		<a href="#">SVE propagate break from previous partition</a>
1		UNALLOCATED

## SVE propagate break from previous partition

These instructions are under [SVE Propagate Break](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	op	S	0	0	Pm				1	1	Pg				0	Pn				B	Pd			

Decode fields			Instruction Details
op	S	B	
0	0	0	<a href="#">BRKPA</a>
0	0	1	<a href="#">BRKPB</a>
0	1	0	<a href="#">BRKPAS</a>
0	1	1	<a href="#">BRKPBS</a>
1			UNALLOCATED

## SVE Partition Break

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
00100101								op0				01		op1				01						op2						op3			

Decode fields				Instruction details
op0	op1	op2	op3	
0	1000	0	0	<a href="#">SVE propagate break to next partition</a>
0	1000	0	1	UNALLOCATED
0	x000	1		UNALLOCATED
0	x1xx			UNALLOCATED
0	xx1x			UNALLOCATED
0	xxx1			UNALLOCATED
1	0000	1		UNALLOCATED
1	!= 0000			UNALLOCATED
	0000	0		<a href="#">SVE partition break condition</a>

## SVE propagate break to next partition

These instructions are under [SVE Partition Break](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	S	0	1	1	0	0	0	0	1	Pg			0	Pn			0	Pdm					

Decode fields	Instruction Details
0	<a href="#">BRKN</a>
1	<a href="#">BRKNS</a>

## SVE partition break condition

These instructions are under [SVE Partition Break](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	B	S	0	1	0	0	0	0	0	1	Pg			0	Pn			M	Pd					

Decode fields			Instruction Details
B	S	M	
	1	1	UNALLOCATED
0	0		<a href="#">BRKA</a>
0	1	0	<a href="#">BRKAS</a>
1	0		<a href="#">BRKB</a>
1	1	0	<a href="#">BRKBS</a>

## SVE Predicate Misc

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00100101										01		op0			11		op1		op2		op3			op4							

Decode fields					Instruction details
op0	op1	op2	op3	op4	
0000		x0		0	<a href="#">SVE predicate test</a>
0100		x0		0	UNALLOCATED
0x10		x0		0	UNALLOCATED
0xx1		x0		0	UNALLOCATED
0xxx		x1		0	UNALLOCATED
1000	000	00		0	<a href="#">SVE predicate first active</a>
1000	000	!= 00		0	UNALLOCATED
1000	100	10	0000	0	<a href="#">SVE predicate zero</a>
1000	100	10	!= 0000	0	UNALLOCATED
1000	110	00		0	<a href="#">SVE predicate read from FFR (predicated)</a>
1001	000	0x		0	UNALLOCATED
1001	000	10		0	<a href="#">PNEXT</a>
1001	000	11		0	UNALLOCATED
1001	100	10		0	UNALLOCATED
1001	110	00	0000	0	<a href="#">SVE predicate read from FFR (unpredicated)</a>
1001	110	00	!= 0000	0	UNALLOCATED
100x	010			0	UNALLOCATED
100x	100	0x		0	<a href="#">SVE predicate initialize</a>
100x	100	11		0	UNALLOCATED
100x	110	!= 00		0	UNALLOCATED
100x	xx1			0	UNALLOCATED
110x				0	UNALLOCATED



1x1x				0	UNALLOCATED
				1	UNALLOCATED

**SVE predicate test**

These instructions are under [SVE Predicate Misc.](#)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	op	S	0	1	0	0	0	0	1	1		Pg		0			Pn		0		opc2			

Decode fields			Instruction Details
op	S	opc2	
0	0		UNALLOCATED
0	1	0000	<a href="#">PTEST</a>
0	1	0001	UNALLOCATED
0	1	001x	UNALLOCATED
0	1	01xx	UNALLOCATED
0	1	1xxx	UNALLOCATED
1			UNALLOCATED

**SVE predicate first active**

These instructions are under [SVE Predicate Misc.](#)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	op	S	0	1	1	0	0	0	1	1	0	0	0	0	0	0		Pg		0		Pdn		

Decode fields		Instruction Details
op	S	
0	0	UNALLOCATED
0	1	<a href="#">PFIRST</a>
1		UNALLOCATED

**SVE predicate zero**

These instructions are under [SVE Predicate Misc.](#)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	op	S	0	1	1	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	0	Pd		

Decode fields		Instruction Details
op	S	
0	0	<a href="#">PFALSE</a>
0	1	UNALLOCATED
1		UNALLOCATED

**SVE predicate read from FFR (predicated)**

These instructions are under [SVE Predicate Misc.](#)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	op	S	0	1	1	0	0	0	1	1	1	1	0	0	0		Pg		0		Pd			

Decode fields op	S	Instruction Details
0	0	<a href="#">RDFFR (predicated)</a>
0	1	<a href="#">RDFFRS</a>
1		UNALLOCATED

**SVE predicate read from FFR (unpredicated)**

These instructions are under [SVE Predicate Misc.](#)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	op	S	0	1	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0				Pd

Decode fields op	S	Instruction Details
0	0	<a href="#">RDFFR (unpredicated)</a>
0	1	UNALLOCATED
1		UNALLOCATED

**SVE predicate initialize**

These instructions are under [SVE Predicate Misc.](#)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size		0	1	1	0	0	S	1	1	1	0	0	0				pattern		0				Pd

Decode fields S	Instruction Details
0	<a href="#">PTRUE (predicate)</a>
1	<a href="#">PTRUES</a>

**SVE Integer Compare - Scalars**

These instructions are under [SVE encodings.](#)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										1						00		op0		op1											op2

op0	Decode fields op1	op2	Instruction details
0x			<a href="#">SVE integer compare scalar count and limit</a>
10	00	0000	<a href="#">SVE conditionally terminate scalars</a>
10	00	!= 0000	UNALLOCATED
11	00		<a href="#">SVE pointer conflict compare</a>
1x	!= 00		UNALLOCATED

**SVE integer compare scalar count and limit**

These instructions are under [SVE Integer Compare - Scalars.](#)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	Rm			0	0	0	sf	U	lt	Rn			eq	Pd								

Decode fields			Instruction Details
U	lt	eq	
0	0	0	<a href="#">WHILEGE (predicate)</a>
0	0	1	<a href="#">WHILEGT (predicate)</a>
0	1	0	<a href="#">WHILELT (predicate)</a>
0	1	1	<a href="#">WHILELE (predicate)</a>
1	0	0	<a href="#">WHILEHS (predicate)</a>
1	0	1	<a href="#">WHILEHI (predicate)</a>
1	1	0	<a href="#">WHILELO (predicate)</a>
1	1	1	<a href="#">WHILELS (predicate)</a>

### SVE conditionally terminate scalars

These instructions are under [SVE Integer Compare - Scalars](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	op	sz	1	Rm					0	0	1	0	0	0	Rn					ne	0	0	0	0

Decode fields		Instruction Details
op	ne	
0		UNALLOCATED
1	0	<a href="#">CTERMEQ, CTERMNE — CTERMEQ</a>
1	1	<a href="#">CTERMEQ, CTERMNE — CTERMNE</a>

### SVE pointer conflict compare

These instructions are under [SVE Integer Compare - Scalars](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	0	1	0	1	size	1	Rm						0	0	1	1	0	0	Rn						rw	Pd			

Decode fields		Instruction Details
rw		
0		<a href="#">WHILEWR</a>
1		<a href="#">WHILERW</a>

### SVE broadcast predicate element

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	i1	tszh	1		tszl		Rv		0	1		Pn		S			Pm		0			Pd		

Decode fields		Instruction Details	Feature
S			
0		<a href="#">PSEL</a>	FEAT_SVE2p1
1		UNALLOCATED	-

### SVE Scalar Integer Compare - Predicate-as-counter

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00100101									1	op0				01	op1		op2				1	op3									

Decode fields				Instruction details	Feature
op0	op1	op2	op3		
00000	110			<a href="#">SVE extract mask predicate from predicate-as-counter</a>	-
00000	111	000000	0	<a href="#">PTRUE (predicate as counter)</a>	FEAT_SVE2p1
00000	111	000000	1	UNALLOCATED	-
00000	111	!= 000000		UNALLOCATED	-
!= 00000	11x			UNALLOCATED	-
	01x			<a href="#">SVE integer compare scalar count and limit (predicate pair)</a>	-
	x0x			<a href="#">SVE integer compare scalar count and limit (predicate-as-counter)</a>	-

**SVE extract mask predicate from predicate-as-counter**

These instructions are under [SVE Scalar Integer Compare - Predicate-as-counter](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size		1	0	0	0	0	0	0	1	1	1	0	opc			Pn		1			Pd		

Decode fields opc	Instruction Details	Feature
0xx	<a href="#">PEXT (predicate)</a>	FEAT_SVE2p1
10x	<a href="#">PEXT (predicate pair)</a>	FEAT_SVE2p1
11x	UNALLOCATED	-

**SVE integer compare scalar count and limit (predicate pair)**

These instructions are under [SVE Scalar Integer Compare - Predicate-as-counter](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	Rm				0	1	0	1	U	lt	Rn				1	Pd		eq				

Decode fields U lt eq			Instruction Details	Feature
U	lt	eq		
0	0	0	<a href="#">WHILEGE (predicate pair)</a>	FEAT_SVE2p1
0	0	1	<a href="#">WHILEGT (predicate pair)</a>	FEAT_SVE2p1
0	1	0	<a href="#">WHILELT (predicate pair)</a>	FEAT_SVE2p1
0	1	1	<a href="#">WHILELE (predicate pair)</a>	FEAT_SVE2p1
1	0	0	<a href="#">WHILEHS (predicate pair)</a>	FEAT_SVE2p1
1	0	1	<a href="#">WHILEHI (predicate pair)</a>	FEAT_SVE2p1
1	1	0	<a href="#">WHILELO (predicate pair)</a>	FEAT_SVE2p1
1	1	1	<a href="#">WHILELS (predicate pair)</a>	FEAT_SVE2p1

**SVE integer compare scalar count and limit (predicate-as-counter)**

These instructions are under [SVE Scalar Integer Compare - Predicate-as-counter](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size		1			Rm			0	1	vi	0	U	lt			Rn		1	eq		Pnd		

Decode fields			Instruction Details	Feature
U	lt	eq		
0	0	0	<a href="#">WHILEGE (predicate as counter)</a>	FEAT_SVE2p1

Decode fields			Instruction Details	Feature
U	It	eq		
0	0	1	<a href="#">WHILEGT (predicate as counter)</a>	FEAT_SVE2p1
0	1	0	<a href="#">WHILELT (predicate as counter)</a>	FEAT_SVE2p1
0	1	1	<a href="#">WHILELE (predicate as counter)</a>	FEAT_SVE2p1
1	0	0	<a href="#">WHILEHS (predicate as counter)</a>	FEAT_SVE2p1
1	0	1	<a href="#">WHILEHI (predicate as counter)</a>	FEAT_SVE2p1
1	1	0	<a href="#">WHILELO (predicate as counter)</a>	FEAT_SVE2p1
1	1	1	<a href="#">WHILELS (predicate as counter)</a>	FEAT_SVE2p1

## SVE Integer Wide Immediate - Unpredicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00100101										1	op0			op1	11																

Decode fields		Instruction details
op0	op1	
00		<a href="#">SVE integer add/subtract immediate (unpredicated)</a>
01		<a href="#">SVE integer min/max immediate (unpredicated)</a>
10		<a href="#">SVE integer multiply immediate (unpredicated)</a>
11	0	<a href="#">SVE broadcast integer immediate (unpredicated)</a>
11	1	<a href="#">SVE broadcast floating-point immediate (unpredicated)</a>

## SVE integer add/subtract immediate (unpredicated)

These instructions are under [SVE Integer Wide Immediate - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size		1	0	0	opc		1		1	sh	imm8								Zdn				

Decode fields	Instruction Details
opc	
000	<a href="#">ADD (immediate)</a>
001	<a href="#">SUB (immediate)</a>
010	UNALLOCATED
011	<a href="#">SUBR (immediate)</a>
100	<a href="#">SQADD (immediate)</a>
101	<a href="#">UQADD (immediate)</a>
110	<a href="#">SQSUB (immediate)</a>
111	<a href="#">UQSUB (immediate)</a>

## SVE integer min/max immediate (unpredicated)

These instructions are under [SVE Integer Wide Immediate - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	0	1	opc	1	1	o2	imm8								Zdn							

Decode fields	Instruction Details
opc	
0xx	UNALLOCATED
000	<a href="#">SMAX (immediate)</a>

Decode fields opc	o2	Instruction Details
001	0	<a href="#">UMAX (immediate)</a>
010	0	<a href="#">SMIN (immediate)</a>
011	0	<a href="#">UMIN (immediate)</a>
1xx		UNALLOCATED

**SVE integer multiply immediate (unpredicated)**

These instructions are under [SVE Integer Wide Immediate - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size		1	1	0	opc		1		1	o2	imm8								Zdn				

Decode fields opc	o2	Instruction Details
000	0	<a href="#">MUL (immediate)</a>
000	1	UNALLOCATED
001		UNALLOCATED
01x		UNALLOCATED
1xx		UNALLOCATED

**SVE broadcast integer immediate (unpredicated)**

These instructions are under [SVE Integer Wide Immediate - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	1	1	opc	0	1	1	sh	imm8								Zd						

Decode fields opc	Instruction Details
00	<a href="#">DUP (immediate)</a>
01	UNALLOCATED
1x	UNALLOCATED

**SVE broadcast floating-point immediate (unpredicated)**

These instructions are under [SVE Integer Wide Immediate - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	1	1	opc	1	1	1	o2	imm8								Zd						

Decode fields opc	o2	Instruction Details
00	0	<a href="#">FDUP</a>
00	1	UNALLOCATED
01		UNALLOCATED
1x		UNALLOCATED

**SVE Predicate Count**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00100101										100					10		op0		op1												

Decode fields		Instruction details
op0	op1	
000	1	<a href="#">SVE predicate count (predicate-as-counter)</a>
!= 000	1	UNALLOCATED
	0	<a href="#">SVE predicate count</a>

### SVE predicate count (predicate-as-counter)

These instructions are under [SVE Predicate Count](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	0	0	opc	1	0	0	0	0	0	0	vl	1			Pn					Rd		

Decode fields		Instruction Details	Feature
opc			
000		<a href="#">CNTp (predicate as counter)</a>	FEAT_SVE2p1
001		UNALLOCATED	-
01x		UNALLOCATED	-
1xx		UNALLOCATED	-

### SVE predicate count

These instructions are under [SVE Predicate Count](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	0	0	opc	1	0					Pg		0			Pn					Rd		

Decode fields		Instruction Details
opc		
000		<a href="#">CNTp (predicate)</a>
001		UNALLOCATED
01x		UNALLOCATED
1xx		UNALLOCATED

### SVE Inc/Dec by Predicate Count

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
												101	op0																		

Decode fields		Instruction details
op0	op1	
0	0	<a href="#">SVE saturating inc/dec vector by predicate count</a>
0	1	<a href="#">SVE saturating inc/dec register by predicate count</a>
1	0	<a href="#">SVE inc/dec vector by predicate count</a>
1	1	<a href="#">SVE inc/dec register by predicate count</a>

### SVE saturating inc/dec vector by predicate count

These instructions are under [SVE Inc/Dec by Predicate Count](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size		1	0	1	0	D	U	1	0	0	0	0	opc		Pm			Zdn					

Decode fields			Instruction Details
D	U	opc	
		01	UNALLOCATED
		1x	UNALLOCATED
0	0	00	<a href="#">SQINCP (vector)</a>
0	1	00	<a href="#">UQINCP (vector)</a>
1	0	00	<a href="#">SQDECP (vector)</a>
1	1	00	<a href="#">UQDECP (vector)</a>

**SVE saturating inc/dec register by predicate count**

These instructions are under [SVE Inc/Dec by Predicate Count](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	0	1	0	D	U	1	0	0	0	1	sf	op	Pm			Rdn						

Decode fields				Instruction Details
D	U	sf	op	
			1	UNALLOCATED
0	0	0	0	<a href="#">SQINCP (scalar) — 32-bit</a>
0	0	1	0	<a href="#">SQINCP (scalar) — 64-bit</a>
0	1	0	0	<a href="#">UQINCP (scalar) — 32-bit</a>
0	1	1	0	<a href="#">UQINCP (scalar) — 64-bit</a>
1	0	0	0	<a href="#">SQDECP (scalar) — 32-bit</a>
1	0	1	0	<a href="#">SQDECP (scalar) — 64-bit</a>
1	1	0	0	<a href="#">UQDECP (scalar) — 32-bit</a>
1	1	1	0	<a href="#">UQDECP (scalar) — 64-bit</a>

**SVE inc/dec vector by predicate count**

These instructions are under [SVE Inc/Dec by Predicate Count](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size		1	0	1	1	op	D	1	0	0	0	0	opc2		Pm			Zdn					

Decode fields			Instruction Details
op	D	opc2	
0		01	UNALLOCATED
0		1x	UNALLOCATED
0	0	00	<a href="#">INCP (vector)</a>
0	1	00	<a href="#">DECP (vector)</a>
1			UNALLOCATED

**SVE inc/dec register by predicate count**

These instructions are under [SVE Inc/Dec by Predicate Count](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size	1	0	1	1	op	D	1	0	0	0	1	opc2	Pm			Rdn							

Decode fields			Instruction Details
op	D	opc2	
0		01	UNALLOCATED
0		1x	UNALLOCATED



Decode fields			Instruction Details
op	D	opc2	
0	0	00	<a href="#">INCP (scalar)</a>
0	1	00	<a href="#">DECP (scalar)</a>
1			UNALLOCATED

## SVE Write FFR

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00100101										101		op0	op1			1001				op2				op3						op4	

Decode fields					Instruction details
op0	op1	op2	op3	op4	
0	00	000		00000	<a href="#">SVE FFR write from predicate</a>
1	00	000	0000	00000	<a href="#">SVE FFR initialise</a>
1	00	000	1xxx	00000	UNALLOCATED
1	00	000	x1xx	00000	UNALLOCATED
1	00	000	xx1x	00000	UNALLOCATED
1	00	000	xxx1	00000	UNALLOCATED
	00	000		!= 00000	UNALLOCATED
	00	!= 000			UNALLOCATED
	!= 00				UNALLOCATED

## SVE FFR write from predicate

These instructions are under [SVE Write FFR](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	opc		1	0	1	0	0	0	1	0	0	1	0	0	0	0		Pn		0	0	0	0	0

Decode fields	Instruction Details
opc	
00	<a href="#">WRFFR</a>
01	UNALLOCATED
1x	UNALLOCATED

## SVE FFR initialise

These instructions are under [SVE Write FFR](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	opc		1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

Decode fields	Instruction Details
opc	
00	<a href="#">SETFFR</a>
01	UNALLOCATED
1x	UNALLOCATED

## SVE Integer Multiply-Add - Unpredicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01000100										0						0															

### Decode fields op0

### Instruction details

0000x	<a href="#">SVE integer dot product (unpredicated)</a>
0001x	<a href="#">SVE2 saturating multiply-add interleaved long</a>
001xx	<a href="#">CDOT (vectors)</a>
01xxx	<a href="#">SVE2 complex integer multiply-add</a>
10xxx	<a href="#">SVE2 integer multiply-add long</a>
110xx	<a href="#">SVE2 saturating multiply-add long</a>
1110x	<a href="#">SVE2 saturating multiply-add high</a>
11110	<a href="#">SVE mixed sign dot product</a>
11111	UNALLOCATED

## SVE integer dot product (unpredicated)

These instructions are under [SVE Integer Multiply-Add - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	0	Zm			0	0	0	0	0	U	Zn			Zda									

### Decode fields U

### Instruction Details

0	<a href="#">SDOT (4-way, vectors)</a>
1	<a href="#">UDOT (4-way, vectors)</a>

## SVE2 saturating multiply-add interleaved long

These instructions are under [SVE Integer Multiply-Add - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	0	0	0	1	0	0	size	0	Zm						0	0	0	0	1	S	Zn						Zda					

### Decode fields S

### Instruction Details

0	<a href="#">SQDMLALBT</a>
1	<a href="#">SQDMLSLBT</a>

## SVE2 complex integer multiply-add

These instructions are under [SVE Integer Multiply-Add - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	0	Zm			0	0	1	op	rot	Zn			Zda										

### Decode fields op

### Instruction Details

0	<a href="#">CMLA (vectors)</a>
1	<a href="#">SQRDCMLAH (vectors)</a>

## SVE2 integer multiply-add long

These instructions are under [SVE Integer Multiply-Add - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	0			Zm				0	1	0	S	U	T				Zn				Zda		

Decode fields			Instruction Details
S	U	T	
0	0	0	<a href="#">SMLALB (vectors)</a>
0	0	1	<a href="#">SMLALT (vectors)</a>
0	1	0	<a href="#">UMLALB (vectors)</a>
0	1	1	<a href="#">UMLALT (vectors)</a>
1	0	0	<a href="#">SMLSLB (vectors)</a>
1	0	1	<a href="#">SMLSLT (vectors)</a>
1	1	0	<a href="#">UMLSLB (vectors)</a>
1	1	1	<a href="#">UMLSLT (vectors)</a>

SVE2 saturating multiply-add long

These instructions are under [SVE Integer Multiply-Add - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	0	0	0	1	0	0	size	0	Zm						0	1	1	0	S	T	Zn						Zda					

Decode fields		Instruction Details
S	T	
0	0	<a href="#">SQDMLALB (vectors)</a>
0	1	<a href="#">SQDMLALT (vectors)</a>
1	0	<a href="#">SQDMLSLB (vectors)</a>
1	1	<a href="#">SQDMLSLT (vectors)</a>

SVE2 saturating multiply-add high

These instructions are under [SVE Integer Multiply-Add - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	0	0	0	1	0	0	size	0	Zm						0	1	1	1	0	S	Zn						Zda					

Decode fields		Instruction Details
S		
0		<a href="#">SQRDMLAH (vectors)</a>
1		<a href="#">SQRDMLSH (vectors)</a>

SVE mixed sign dot product

These instructions are under [SVE Integer Multiply-Add - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	0	Zm						0	1	1	1	1	0	Zn					Zda				

Decode fields	Instruction Details	Feature
size		
0x	UNALLOCATED	-
10	<a href="#">USDOT (vectors)</a>	FEAT_I8MM
11	UNALLOCATED	-

SVE2 Integer - Predicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01000100										0	op0					10	op1														

Decode fields		Instruction details
op0	op1	
0010	1	<a href="#">SVE2 integer pairwise add and accumulate long</a>
0011	1	UNALLOCATED
011x	1	UNALLOCATED
0x0x	1	<a href="#">SVE2 integer unary operations (predicated)</a>
0xxx	0	<a href="#">SVE2 saturating/rounding bitwise shift left (predicated)</a>
10xx	0	<a href="#">SVE2 integer halving add/subtract (predicated)</a>
10xx	1	<a href="#">SVE2 integer pairwise arithmetic</a>
11xx	0	<a href="#">SVE2 saturating add/subtract</a>
11xx	1	UNALLOCATED

**SVE2 integer pairwise add and accumulate long**

These instructions are under [SVE2 Integer - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		0	0	0	1	0	U	1	0	1		Pg					Zn				Zda		

Decode fields		Instruction Details
U		
0		<a href="#">SADALP</a>
1		<a href="#">UADALP</a>

**SVE2 integer unary operations (predicated)**

These instructions are under [SVE2 Integer - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		0	0	Q	0	opc		1	0	1		Pg					Zn				Zd		

Decode fields		Instruction Details
Q	opc	
	1x	UNALLOCATED
0	00	<a href="#">URECPE</a>
0	01	<a href="#">URSQRTE</a>
1	00	<a href="#">SQABS</a>
1	01	<a href="#">SQNEG</a>

**SVE2 saturating/rounding bitwise shift left (predicated)**

These instructions are under [SVE2 Integer - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		0	0	Q	R	N	U	1	0	0		Pg					Zm				Zdn		

Decode fields				Instruction Details
Q	R	N	U	
0		0		UNALLOCATED
0	0	1	0	<a href="#">SRSHL</a>
0	0	1	1	<a href="#">URSHL</a>
0	1	1	0	<a href="#">SRSHLR</a>

Decode fields				Instruction Details
Q	R	N	U	
0	1	1	1	<a href="#">URSHLR</a>
1	0	0	0	<a href="#">SQSHL (vectors)</a>
1	0	0	1	<a href="#">UQSHL (vectors)</a>
1	0	1	0	<a href="#">SQRSHL</a>
1	0	1	1	<a href="#">UQRSHL</a>
1	1	0	0	<a href="#">SQSHLR</a>
1	1	0	1	<a href="#">UQSHLR</a>
1	1	1	0	<a href="#">SQRSHLR</a>
1	1	1	1	<a href="#">UQRSHLR</a>

**SVE2 integer halving add/subtract (predicated)**

These instructions are under [SVE2 Integer - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	0	1	0	R	S	U	1	0	0	Pg			Zm				Zdn						

Decode fields			Instruction Details
R	S	U	
0	0	0	<a href="#">SHADD</a>
0	0	1	<a href="#">UHADD</a>
0	1	0	<a href="#">SHSUB</a>
0	1	1	<a href="#">UHSUB</a>
1	0	0	<a href="#">SRHADD</a>
1	0	1	<a href="#">URHADD</a>
1	1	0	<a href="#">SHSUBR</a>
1	1	1	<a href="#">UHSUBR</a>

**SVE2 integer pairwise arithmetic**

These instructions are under [SVE2 Integer - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	0	1	0	opc	U	1	0	1	Pg		Zm				Zdn								

Decode fields		Instruction Details
opc	U	
00	0	UNALLOCATED
00	1	<a href="#">ADDP</a>
01		UNALLOCATED
10	0	<a href="#">SMAXP</a>
10	1	<a href="#">UMAXP</a>
11	0	<a href="#">SMINP</a>
11	1	<a href="#">UMINP</a>

**SVE2 saturating add/subtract**

These instructions are under [SVE2 Integer - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	0	1	1	op	S	U	1	0	0	Pg	Zm				Zdn								

Decode fields			Instruction Details
op	S	U	
0	0	0	<a href="#">SQADD (vectors, predicated)</a>
0	0	1	<a href="#">UQADD (vectors, predicated)</a>
0	1	0	<a href="#">SQSUB (vectors, predicated)</a>
0	1	1	<a href="#">UQSUB (vectors, predicated)</a>
1	0	0	<a href="#">SUQADD</a>
1	0	1	<a href="#">USQADD</a>
1	1	0	<a href="#">SQSUBR</a>
1	1	1	<a href="#">UQSUBR</a>

## SVE integer clamp

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	0	Zm						1	1	0	0	0	U	Zn						Zd			

Decode fields		Instruction Details	Feature
U			
0		<a href="#">SCLAMP</a>	FEAT_SVE2p1
1		<a href="#">UCLAMP</a>	FEAT_SVE2p1

## SVE permute vector elements (quadwords)

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	0	Zm						1	1	1	opc			Zn					Zd				

Decode fields		Instruction Details	Feature
opc			
000		<a href="#">ZIPQ1</a>	FEAT_SVE2p1
001		<a href="#">ZIPQ2</a>	FEAT_SVE2p1
010		<a href="#">UZPQ1</a>	FEAT_SVE2p1
011		<a href="#">UZPQ2</a>	FEAT_SVE2p1
10x		UNALLOCATED	-
110		<a href="#">TBLQ</a>	FEAT_SVE2p1
111		UNALLOCATED	-

## SVE Multiply - Indexed

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					01000100					1										op0											

Decode fields		Instruction details
op0		
00000x		<a href="#">SVE integer dot product (indexed)</a>
00001x		<a href="#">SVE2 integer multiply-add (indexed)</a>
00010x		<a href="#">SVE2 saturating multiply-add high (indexed)</a>
00011x		<a href="#">SVE mixed sign dot product (indexed)</a>
001xxx		<a href="#">SVE2 saturating multiply-add (indexed)</a>

0100xx	<a href="#">SVE2 complex integer dot product (indexed)</a>
0101xx	UNALLOCATED
0110xx	<a href="#">SVE2 complex integer multiply-add (indexed)</a>
0111xx	<a href="#">SVE2 complex saturating multiply-add (indexed)</a>
10xxxx	<a href="#">SVE2 integer multiply-add long (indexed)</a>
110xxx	<a href="#">SVE2 integer multiply long (indexed)</a>
1110xx	<a href="#">SVE2 saturating multiply (indexed)</a>
11110x	<a href="#">SVE2 saturating multiply high (indexed)</a>
111110	<a href="#">SVE2 integer multiply (indexed)</a>
111111	UNALLOCATED

**SVE integer dot product (indexed)**

These instructions are under [SVE Multiply - Indexed](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	0	0	0	1	0	0	size	1	opc						0	0	0	0	0	U	Zn						Zda					

Decode fields size	U	Instruction Details
0x		UNALLOCATED
10	0	<a href="#">SDOT (4-way, indexed) — 32-bit</a>
10	1	<a href="#">UDOT (4-way, indexed) — 32-bit</a>
11	0	<a href="#">SDOT (4-way, indexed) — 64-bit</a>
11	1	<a href="#">UDOT (4-way, indexed) — 64-bit</a>

**SVE2 integer multiply-add (indexed)**

These instructions are under [SVE Multiply - Indexed](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	1	opc			0	0	0	0	1	S	Zn					Zda							

Decode fields size	S	Instruction Details
0x	0	<a href="#">MLA (indexed) — 16-bit</a>
0x	1	<a href="#">MLS (indexed) — 16-bit</a>
10	0	<a href="#">MLA (indexed) — 32-bit</a>
10	1	<a href="#">MLS (indexed) — 32-bit</a>
11	0	<a href="#">MLA (indexed) — 64-bit</a>
11	1	<a href="#">MLS (indexed) — 64-bit</a>

**SVE2 saturating multiply-add high (indexed)**

These instructions are under [SVE Multiply - Indexed](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	1	opc			0	0	0	1	0	S	Zn					Zda							

Decode fields size	S	Instruction Details
0x	0	<a href="#">SQRDMLAH (indexed) — 16-bit</a>
0x	1	<a href="#">SQRDMLSH (indexed) — 16-bit</a>

Decode fields size	S	Instruction Details
10	0	<a href="#">SQRDMLAH (indexed)</a> — 32-bit
10	1	<a href="#">SQRDMLSH (indexed)</a> — 32-bit
11	0	<a href="#">SQRDMLAH (indexed)</a> — 64-bit
11	1	<a href="#">SQRDMLSH (indexed)</a> — 64-bit

**SVE mixed sign dot product (indexed)**

These instructions are under [SVE Multiply - Indexed](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	1	opc			0	0	0	1	1	U	Zn				Zda								

Decode fields size	U	Instruction Details	Feature
0x		UNALLOCATED	-
10	0	<a href="#">USDOT (indexed)</a>	FEAT_I8MM
10	1	<a href="#">SUDOT</a>	FEAT_I8MM
11		UNALLOCATED	-

**SVE2 saturating multiply-add (indexed)**

These instructions are under [SVE Multiply - Indexed](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	1		opc		0	0	1	S	il	T													

Decode fields size	S	T	Instruction Details
0x			UNALLOCATED
10	0	0	<a href="#">SQDMLALB (indexed)</a> — 32-bit
10	0	1	<a href="#">SQDMLALT (indexed)</a> — 32-bit
10	1	0	<a href="#">SQDMLSLB (indexed)</a> — 32-bit
10	1	1	<a href="#">SQDMLSLT (indexed)</a> — 32-bit
11	0	0	<a href="#">SQDMLALB (indexed)</a> — 64-bit
11	0	1	<a href="#">SQDMLALT (indexed)</a> — 64-bit
11	1	0	<a href="#">SQDMLSLB (indexed)</a> — 64-bit
11	1	1	<a href="#">SQDMLSLT (indexed)</a> — 64-bit

**SVE2 complex integer dot product (indexed)**

These instructions are under [SVE Multiply - Indexed](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	1		opc		0	1	0	0	rot														

Decode fields size	Instruction Details
0x	UNALLOCATED
10	<a href="#">CDOT (indexed)</a> — 32-bit
11	<a href="#">CDOT (indexed)</a> — 64-bit



**SVE2 complex integer multiply-add (indexed)**

These instructions are under [SVE Multiply - Indexed](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	1		opc					0	1	1	0	rot			Zn					Zda			

Decode fields size	Instruction Details
0x	UNALLOCATED
10	<a href="#">CMLA (indexed)</a> — 16-bit
11	<a href="#">CMLA (indexed)</a> — 32-bit

**SVE2 complex saturating multiply-add (indexed)**

These instructions are under [SVE Multiply - Indexed](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	1		opc					0	1	1	1	rot			Zn					Zda			

Decode fields size	Instruction Details
0x	UNALLOCATED
10	<a href="#">SQRDCMLAH (indexed)</a> — 16-bit
11	<a href="#">SQRDCMLAH (indexed)</a> — 32-bit

**SVE2 integer multiply-add long (indexed)**

These instructions are under [SVE Multiply - Indexed](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	1		opc					1	0	S	U	il	T			Zn				Zda			

Decode fields size	S	U	T	Instruction Details
0x				UNALLOCATED
10	0	0	0	<a href="#">SMLALB (indexed)</a> — 32-bit
10	0	0	1	<a href="#">SMLALT (indexed)</a> — 32-bit
10	0	1	0	<a href="#">UMLALB (indexed)</a> — 32-bit
10	0	1	1	<a href="#">UMLALT (indexed)</a> — 32-bit
10	1	0	0	<a href="#">SMLSLB (indexed)</a> — 32-bit
10	1	0	1	<a href="#">SMLSLT (indexed)</a> — 32-bit
10	1	1	0	<a href="#">UMLSLB (indexed)</a> — 32-bit
10	1	1	1	<a href="#">UMLSLT (indexed)</a> — 32-bit
11	0	0	0	<a href="#">SMLALB (indexed)</a> — 64-bit
11	0	0	1	<a href="#">SMLALT (indexed)</a> — 64-bit
11	0	1	0	<a href="#">UMLALB (indexed)</a> — 64-bit
11	0	1	1	<a href="#">UMLALT (indexed)</a> — 64-bit
11	1	0	0	<a href="#">SMLSLB (indexed)</a> — 64-bit
11	1	0	1	<a href="#">SMLSLT (indexed)</a> — 64-bit
11	1	1	0	<a href="#">UMLSLB (indexed)</a> — 64-bit
11	1	1	1	<a href="#">UMLSLT (indexed)</a> — 64-bit

**SVE2 integer multiply long (indexed)**

These instructions are under [SVE Multiply - Indexed](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	1		opc		1	1	0	U	il	T													

Decode fields			Instruction Details
size	U	T	
0x			UNALLOCATED
10	0	0	<a href="#">SMULLB (indexed)</a> — 32-bit
10	0	1	<a href="#">SMULLT (indexed)</a> — 32-bit
10	1	0	<a href="#">UMULLB (indexed)</a> — 32-bit
10	1	1	<a href="#">UMULLT (indexed)</a> — 32-bit
11	0	0	<a href="#">SMULLB (indexed)</a> — 64-bit
11	0	1	<a href="#">SMULLT (indexed)</a> — 64-bit
11	1	0	<a href="#">UMULLB (indexed)</a> — 64-bit
11	1	1	<a href="#">UMULLT (indexed)</a> — 64-bit

**SVE2 saturating multiply (indexed)**

These instructions are under [SVE Multiply - Indexed](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	1		opc		1	1	1	0	il	T													

Decode fields		Instruction Details
size	T	
0x		UNALLOCATED
10	0	<a href="#">SQDMULLB (indexed)</a> — 32-bit
10	1	<a href="#">SQDMULLT (indexed)</a> — 32-bit
11	0	<a href="#">SQDMULLB (indexed)</a> — 64-bit
11	1	<a href="#">SQDMULLT (indexed)</a> — 64-bit

**SVE2 saturating multiply high (indexed)**

These instructions are under [SVE Multiply - Indexed](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	1		opc		1	1	1	1	0	R													

Decode fields		Instruction Details
size	R	
0x	0	<a href="#">SQDMULH (indexed)</a> — 16-bit
0x	1	<a href="#">SQRDMULH (indexed)</a> — 16-bit
10	0	<a href="#">SQDMULH (indexed)</a> — 32-bit
10	1	<a href="#">SQRDMULH (indexed)</a> — 32-bit
11	0	<a href="#">SQDMULH (indexed)</a> — 64-bit
11	1	<a href="#">SQRDMULH (indexed)</a> — 64-bit

**SVE2 integer multiply (indexed)**

These instructions are under [SVE Multiply - Indexed](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size	1		opc		1	1	1	1	1	0													

Decode fields size	Instruction Details
0x	<a href="#">MUL (indexed) — 16-bit</a>
10	<a href="#">MUL (indexed) — 32-bit</a>
11	<a href="#">MUL (indexed) — 64-bit</a>

### SVE two-way dot product

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	0	0	0	Zm					1	1	0	0	1	U	Zn					Zda				

Decode fields U	Instruction Details	Feature
0	<a href="#">SDOT (2-way, vectors)</a>	FEAT_SVE2p1
1	<a href="#">UDOT (2-way, vectors)</a>	FEAT_SVE2p1

### SVE two-way dot product (indexed)

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	0	0	opc					1	1	0	0	1	U	Zn					Zda				

Decode fields U	Instruction Details	Feature
0	<a href="#">SDOT (2-way, indexed)</a>	FEAT_SVE2p1
1	<a href="#">UDOT (2-way, indexed)</a>	FEAT_SVE2p1

### SVE2 Widening Integer Arithmetic

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					0	1	0	0	0	1						0		op0													

Decode fields op0	Instruction details
0x	<a href="#">SVE2 integer add/subtract long</a>
10	<a href="#">SVE2 integer add/subtract wide</a>
11	<a href="#">SVE2 integer multiply long</a>

### SVE2 integer add/subtract long

These instructions are under [SVE2 Widening Integer Arithmetic](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size	0	Zm			0	0	op	S	U	T	Zn				Zd								

Decode fields op S U T	Instruction Details
0 0 0 0	<a href="#">SADDLB</a>
0 0 0 1	<a href="#">SADDLT</a>
0 0 1 0	<a href="#">UADDLB</a>

Decode fields				Instruction Details
op	S	U	T	
0	0	1	1	<a href="#">UADDLT</a>
0	1	0	0	<a href="#">SSUBLB</a>
0	1	0	1	<a href="#">SSUBLT</a>
0	1	1	0	<a href="#">USUBLB</a>
0	1	1	1	<a href="#">USUBLT</a>
1	0			UNALLOCATED
1	1	0	0	<a href="#">SABDLB</a>
1	1	0	1	<a href="#">SABDLT</a>
1	1	1	0	<a href="#">UABDLB</a>
1	1	1	1	<a href="#">UABDLT</a>

**SVE2 integer add/subtract wide**

These instructions are under [SVE2 Widening Integer Arithmetic](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size	0			Zm		0	1	0	S	U	T			Zn						Zd			

Decode fields			Instruction Details
S	U	T	
0	0	0	<a href="#">SADDWB</a>
0	0	1	<a href="#">SADDWT</a>
0	1	0	<a href="#">UADDWB</a>
0	1	1	<a href="#">UADDWT</a>
1	0	0	<a href="#">SSUBWB</a>
1	0	1	<a href="#">SSUBWT</a>
1	1	0	<a href="#">USUBWB</a>
1	1	1	<a href="#">USUBWT</a>

**SVE2 integer multiply long**

These instructions are under [SVE2 Widening Integer Arithmetic](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size	0			Zm		0	1	1	op	U	T			Zn						Zd			

Decode fields				Instruction Details	Feature
size	op	U	T		
	0	0	0	<a href="#">SQDMULLB (vectors)</a>	-
	0	0	1	<a href="#">SQDMULLT (vectors)</a>	-
	1	0	0	<a href="#">SMULLB (vectors)</a>	-
	1	0	1	<a href="#">SMULLT (vectors)</a>	-
	1	1	0	<a href="#">UMULLB (vectors)</a>	-
	1	1	1	<a href="#">UMULLT (vectors)</a>	-
!= 00	0	1	0	<a href="#">PMULLB</a> — 16-bit or 64-bit elements	-
!= 00	0	1	1	<a href="#">PMULLT</a> — 16-bit or 64-bit elements	-
00	0	1	0	<a href="#">PMULLB</a> — 128-bit element	FEAT_SVE_PMULL128
00	0	1	1	<a href="#">PMULLT</a> — 128-bit element	FEAT_SVE_PMULL128

## SVE Misc

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01000101								op0		0					10	op1															

Decode fields		Instruction details
op0	op1	
0	10xx	<a href="#">SVE2 bitwise shift left long</a>
1	10xx	UNALLOCATED
	00xx	<a href="#">SVE2 integer add/subtract interleaved long</a>
	010x	<a href="#">SVE2 bitwise exclusive-or interleaved</a>
	0110	<a href="#">SVE integer matrix multiply accumulate</a>
	0111	UNALLOCATED
	11xx	<a href="#">SVE2 bitwise permute</a>

## SVE2 bitwise shift left long

These instructions are under [SVE Misc](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	0	tszh	0	tszl	imm3			1	0	1	0	U	T	Zn				Zd						

Decode fields		Instruction Details
U	T	
0	0	<a href="#">SSHLLB</a>
0	1	<a href="#">SSHLLT</a>
1	0	<a href="#">USHLLB</a>
1	1	<a href="#">USHLLT</a>

## SVE2 integer add/subtract interleaved long

These instructions are under [SVE Misc](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size		0	Zm				1	0	0	0	S	tb	Zn				Zd						

Decode fields		Instruction Details
S	tb	
0	0	<a href="#">SADDLBT</a>
0	1	UNALLOCATED
1	0	<a href="#">SSUBLBT</a>
1	1	<a href="#">SSUBLTB</a>

## SVE2 bitwise exclusive-or interleaved

These instructions are under [SVE Misc](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size		0	Zm				1	0	0	1	0	tb	Zn				Zd						

Decode fields		Instruction Details
tb		
0		<a href="#">EORBT</a>
1		<a href="#">EORTB</a>

**SVE integer matrix multiply accumulate**

These instructions are under [SVE Misc](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	uns	0				Zm			1	0	0	1	1	0			Zn				Zd			

Decode fields uns	Instruction Details	Feature
00	<a href="#">SMMLA</a>	FEAT_I8MM
01	UNALLOCATED	-
10	<a href="#">USMMLA</a>	FEAT_I8MM
11	<a href="#">UMMLA</a>	FEAT_I8MM

**SVE2 bitwise permute**

These instructions are under [SVE Misc](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size	0	Zm						1	0	1	1	opc		Zn						Zd			

Decode fields opc	Instruction Details	Feature
00	<a href="#">BEXT</a>	FEAT_SVE_BitPerm
01	<a href="#">BDEP</a>	FEAT_SVE_BitPerm
10	<a href="#">BGRP</a>	FEAT_SVE_BitPerm
11	UNALLOCATED	-

**SVE2 Accumulate**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										0		op0				11		op1													

Decode fields op0	Decode fields op1	Instruction details
0000	011	<a href="#">SVE2 complex integer add</a>
!= 0000	011	UNALLOCATED
	00x	<a href="#">SVE2 integer absolute difference and accumulate long</a>
	010	<a href="#">SVE2 integer add/subtract long with carry</a>
	10x	<a href="#">SVE2 bitwise shift right and accumulate</a>
	110	<a href="#">SVE2 bitwise shift and insert</a>
	111	<a href="#">SVE2 integer absolute difference and accumulate</a>

**SVE2 complex integer add**

These instructions are under [SVE2 Accumulate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size	0	0	0	0	0	0	op	1	1	0	1	1	rot			Zm				Zdn			

Decode fields op	Instruction Details
0	<a href="#">CADD</a>
1	<a href="#">SQCADD</a>

**SVE2 integer absolute difference and accumulate long**

These instructions are under [SVE2 Accumulate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size	0	Zm						1	1	0	0	U	T	Zn						Zda			

Decode fields		Instruction Details
U	T	
0	0	<a href="#">SABALB</a>
0	1	<a href="#">SABALT</a>
1	0	<a href="#">UABALB</a>
1	1	<a href="#">UABALT</a>

**SVE2 integer add/subtract long with carry**

These instructions are under [SVE2 Accumulate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	0	1	0	1	size	0	Zm						1	1	0	1	0	T	Zn						Zda				

Decode fields		Instruction Details
size	T	
0x	0	<a href="#">ADCLB</a>
0x	1	<a href="#">ADCLT</a>
1x	0	<a href="#">SBCLB</a>
1x	1	<a href="#">SBCLT</a>

**SVE2 bitwise shift right and accumulate**

These instructions are under [SVE2 Accumulate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	tszh	0	tszl			imm3			1	1	1	0	R	U			Zn					Zda		

Decode fields		Instruction Details
R	U	
0	0	<a href="#">SSRA</a>
0	1	<a href="#">USRA</a>
1	0	<a href="#">SRSRA</a>
1	1	<a href="#">URSRA</a>

**SVE2 bitwise shift and insert**

These instructions are under [SVE2 Accumulate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	tszh	0	tszl			imm3			1	1	1	1	0	op			Zn					Zd		

Decode fields	Instruction Details
op	
0	<a href="#">SRI</a>
1	<a href="#">SLI</a>

**SVE2 integer absolute difference and accumulate**

These instructions are under [SVE2 Accumulate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size		0	Zm					1	1	1	1	1	U	Zn					Zda				

Decode fields	Instruction Details
<b>U</b>	
0	<a href="#">SABA</a>
1	<a href="#">UABA</a>

## SVE2 Narrowing

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01000101								op0		1			op1	op2	0		op3			op4					op5						

Decode fields						Instruction details
op0	op1	op2	op3	op4	op5	
0	00	0	10			<a href="#">SVE2 saturating extract narrow</a>
0	00	1	10	0	0	<a href="#">SME2 multi-vec extract narrow</a>
0	00	1	10	0	1	UNALLOCATED
0	00	1	10	1		UNALLOCATED
0	!= 00		10			UNALLOCATED
0			0x			<a href="#">SVE2 bitwise shift right narrow</a>
1			0x	0	0	<a href="#">SME2 multi-vec shift narrow</a>
1			0x	0	1	UNALLOCATED
1			0x	1		UNALLOCATED
1			10			UNALLOCATED
			11			<a href="#">SVE2 integer add/subtract narrow high part</a>

## SVE2 saturating extract narrow

These instructions are under [SVE2 Narrowing](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	0	tszh	1	tszl		0	0	0	0	1	0	opc	T					Zn					Zd	

Decode fields		Instruction Details
opc	T	
00	0	<a href="#">SQXTNB</a>
00	1	<a href="#">SQXTNT</a>
01	0	<a href="#">UQXTNB</a>
01	1	<a href="#">UQXTNT</a>
10	0	<a href="#">SQXTUNB</a>
10	1	<a href="#">SQXTUNT</a>
11		UNALLOCATED

## SME2 multi-vec extract narrow

These instructions are under [SVE2 Narrowing](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	0	tszh	1	tszl		0	0	1	0	1	0	opc	0					Zn		0			Zd	



Decode fields			Instruction Details	Feature
tszh	tszl	opc		
0	0x		UNALLOCATED	-
0	10	00	<a href="#">SQCVTN</a>	FEAT_SVE2p1
0	10	01	<a href="#">UQCVTN</a>	FEAT_SVE2p1
0	10	10	<a href="#">SQCVTUN</a>	FEAT_SVE2p1
0	10	11	UNALLOCATED	-
0	11		UNALLOCATED	-
1			UNALLOCATED	-

**SVE2 bitwise shift right narrow**

These instructions are under [SVE2 Narrowing](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	0	tszh	1	tszl		imm3		0	0	op	U	R	T											
																							Zn						Zd		

Decode fields				Instruction Details
op	U	R	T	
0	0	0	0	<a href="#">SQSHRUNB</a>
0	0	0	1	<a href="#">SQSHRUNT</a>
0	0	1	0	<a href="#">SQRSHRUNB</a>
0	0	1	1	<a href="#">SQRSHRUNT</a>
0	1	0	0	<a href="#">SHRNB</a>
0	1	0	1	<a href="#">SHRNT</a>
0	1	1	0	<a href="#">RSHRNB</a>
0	1	1	1	<a href="#">RSHRNT</a>
1	0	0	0	<a href="#">SQSHRNB</a>
1	0	0	1	<a href="#">SQSHRNT</a>
1	0	1	0	<a href="#">SQRSHRNB</a>
1	0	1	1	<a href="#">SQRSHRNT</a>
1	1	0	0	<a href="#">UQSHRNB</a>
1	1	0	1	<a href="#">UQSHRNT</a>
1	1	1	0	<a href="#">UQRSHRNB</a>
1	1	1	1	<a href="#">UQRSHRNT</a>

**SME2 multi-vec shift narrow**

These instructions are under [SVE2 Narrowing](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	0	1	0	1	1	tszh	1	tszl		imm4				0	0	op	U	R	0		Zn				0		Zd		

Decode fields					Instruction Details	Feature
tszh	tszl	op	U	R		
0	0				UNALLOCATED	-
0	1	0	0	0	UNALLOCATED	-
0	1	0	0	1	<a href="#">SQRSHRUN</a>	FEAT_SVE2p1
0	1	0	1		UNALLOCATED	-
0	1	1		0	UNALLOCATED	-
0	1	1	0	1	<a href="#">SQRSHRN</a>	FEAT_SVE2p1
0	1	1	1	1	<a href="#">UQRSHRN</a>	FEAT_SVE2p1

Decode fields					Instruction Details	Feature
tszh	tszl	op	U	R		
1					UNALLOCATED	-

SVE2 integer add/subtract narrow high part

These instructions are under [SVE2 Narrowing](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size		1	Zm				0	1	1	S	R	T	Zn				Zd						

Decode fields			Instruction Details
S	R	T	
0	0	0	<a href="#">ADDHNB</a>
0	0	1	<a href="#">ADDHNT</a>
0	1	0	<a href="#">RADDHNB</a>
0	1	1	<a href="#">RADDHNT</a>
1	0	0	<a href="#">SUBHNB</a>
1	0	1	<a href="#">SUBHNT</a>
1	1	0	<a href="#">RSUBHNB</a>
1	1	1	<a href="#">RSUBHNT</a>

SVE2 character match

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size		1	Zm				1	0	0	Pg		Zn				op		Pd					

Decode fields		Instruction Details
op		
0		<a href="#">MATCH</a>
1		<a href="#">NMATCH</a>

SVE2 Histogram Computation - Segment

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01000101										1					101		op0														

Decode fields		Instruction details
op0		
000		<a href="#">HISTSEG</a>
!= 000		UNALLOCATED

SVE2 Crypto Extensions

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01000101									1	op0		op1		111		op2			op3												

Decode fields				Instruction details
op0	op1	op2	op3	

000	00	00	00000	<a href="#">SVE2 crypto unary operations</a>
000	00	00	!= 00000	UNALLOCATED
000	00	x1		UNALLOCATED
000	01	0x		UNALLOCATED
000	01	11		UNALLOCATED
000	1x	00		<a href="#">SVE2 crypto destructive binary operations</a>
000	1x	x1		UNALLOCATED
!= 000		0x		UNALLOCATED
!= 000		11		UNALLOCATED
		10		<a href="#">SVE2 crypto constructive binary operations</a>

### SVE2 crypto unary operations

These instructions are under [SVE2 Crypto Extensions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size		1	0	0	0	0	0	1	1	1	0	0	op	0	0	0	0	0	Zdn				

Decode fields size	op	Instruction Details	Feature
00	0	<a href="#">AESMC</a>	FEAT_SVE_AES
00	1	<a href="#">AESIMC</a>	FEAT_SVE_AES
01		UNALLOCATED	-
1x		UNALLOCATED	-

### SVE2 crypto destructive binary operations

These instructions are under [SVE2 Crypto Extensions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size		1	0	0	0	1	op	1	1	1	0	0	o2	Zm					Zdn				

Decode fields size	op	o2	Instruction Details	Feature
00	0	0	<a href="#">AESE</a>	FEAT_SVE_AES
00	0	1	<a href="#">AESD</a>	FEAT_SVE_AES
00	1	0	<a href="#">SM4E</a>	FEAT_SVE_SM4
00	1	1	UNALLOCATED	-
01			UNALLOCATED	-
1x			UNALLOCATED	-

### SVE2 crypto constructive binary operations

These instructions are under [SVE2 Crypto Extensions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size	1	Zm			1	1	1	1	0	op	Zn			Zd									

Decode fields size	op	Instruction Details	Feature
00	0	<a href="#">SM4EKEY</a>	FEAT_SVE_SM4
00	1	<a href="#">RAX1</a>	FEAT_SVE_SHA3
01		UNALLOCATED	-

Decode fields size op	Instruction Details	Feature
1x	UNALLOCATED	-

### SVE floating-point convert precision odd elements

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	opc	0	0	1	0	opc2	1	0	1	Pg	Zn	Zd												

Decode fields opc op2	Instruction Details	Feature
x0	11	UNALLOCATED
00	0x	UNALLOCATED
00	10	<a href="#">FCVTXNT</a>
01		UNALLOCATED
10	00	<a href="#">FCVTNT</a> — <a href="#">single-precision to half-precision</a>
10	01	<a href="#">FCVTLT</a> — <a href="#">half-precision to single-precision</a>
10	10	<a href="#">BFCVTNT</a>
11	0x	UNALLOCATED
11	10	<a href="#">FCVTNT</a> — <a href="#">double-precision to single-precision</a>
11	11	<a href="#">FCVTLT</a> — <a href="#">single-precision to double-precision</a>

### SVE2 floating-point pairwise operations

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	size	0	1	0	opc	1	0	0	Pg	Zm	Zdn													

Decode fields opc	Instruction Details
000	<a href="#">FADDP</a>
001	UNALLOCATED
01x	UNALLOCATED
100	<a href="#">FMAXNMP</a>
101	<a href="#">FMINNMP</a>
110	<a href="#">FMAXP</a>
111	<a href="#">FMINP</a>

### SVE floating-point recursive reduction (quadwords)

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	size	0	1	0	opc	1	0	1	Pg	Zn	Vd													

Decode fields opc	Instruction Details	Feature
000	<a href="#">FADDQV</a>	FEAT_SVE2p1
001	UNALLOCATED	-
01x	UNALLOCATED	-
100	<a href="#">FMAXNMQV</a>	FEAT_SVE2p1

Decode fields opc	Instruction Details	Feature
101	<a href="#">FMINNMQV</a>	FEAT_SVE2p1
110	<a href="#">FMAXQV</a>	FEAT_SVE2p1
111	<a href="#">FMINQV</a>	FEAT_SVE2p1

**SVE floating-point multiply-add (indexed)**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	size	1	opc				0	0	0	0	o2	op	Zn				Zda							

Decode fields size	o2	op	Instruction Details	Feature
0x	0	0	<a href="#">FMLA (indexed)</a> — half-precision	-
0x	0	1	<a href="#">FMLS (indexed)</a> — half-precision	-
0x	1	0	<a href="#">BFMLA (indexed)</a>	FEAT_SVE_B16B16
0x	1	1	<a href="#">BFMLS (indexed)</a>	FEAT_SVE_B16B16
1x	1		UNALLOCATED	-
10	0	0	<a href="#">FMLA (indexed)</a> — single-precision	-
10	0	1	<a href="#">FMLS (indexed)</a> — single-precision	-
11	0	0	<a href="#">FMLA (indexed)</a> — double-precision	-
11	0	1	<a href="#">FMLS (indexed)</a> — double-precision	-

**SVE floating-point complex multiply-add (indexed)**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	size	1	opc				0	0	0	1	rot		Zn				Zda							

Decode fields size	Instruction Details
0x	UNALLOCATED
10	<a href="#">FCMLA (indexed)</a> — half-precision
11	<a href="#">FCMLA (indexed)</a> — single-precision

**SVE floating-point multiply (indexed)**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	size	1	opc			0	0	1	0	o2	0	Zn			Zd									

Decode fields size	o2	Instruction Details	Feature
0x	0	<a href="#">FMUL (indexed)</a> — half-precision	-
0x	1	<a href="#">BFMUL (indexed)</a>	FEAT_SVE_B16B16
1x	1	UNALLOCATED	-
10	0	<a href="#">FMUL (indexed)</a> — single-precision	-
11	0	<a href="#">FMUL (indexed)</a> — double-precision	-

**SVE FP clamp**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	1	0	0	1	0	0	size	1	Zm						0	0	1	0	0	1	Zn						Zd				

Decode fields size	Instruction Details	Feature
!= 00	<a href="#">FCLAMP</a>	FEAT_SVE2p1
00	<a href="#">BFCLAMP</a>	FEAT_SVE_B16B16

**SVE Floating Point Widening Multiply-Add - Indexed**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					01100100			op0		1						01	op1	0													

Decode fields op0	Decode fields op1	Instruction details
0	0	<a href="#">SVE BFloat16 floating-point dot product (indexed)</a>
0	1	UNALLOCATED
1		<a href="#">SVE floating-point multiply-add long (indexed)</a>

**SVE BFloat16 floating-point dot product (indexed)**

These instructions are under [SVE Floating Point Widening Multiply-Add - Indexed](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	0	op	1	i2		Zm			0	1	0	0	opc2			Zn					Zda			

Decode fields op	Decode fields opc2	Instruction Details	Feature
	01	UNALLOCATED	-
	1x	UNALLOCATED	-
0	00	<a href="#">FDOT (indexed)</a>	FEAT_SVE2p1
1	00	<a href="#">BFDOT (indexed)</a>	FEAT_BF16

**SVE floating-point multiply-add long (indexed)**

These instructions are under [SVE Floating Point Widening Multiply-Add - Indexed](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	o2	1	i3h		Zm			0	1	op	0	i3l	T			Zn				Zda			

Decode fields o2	Decode fields op	Decode fields T	Instruction Details	Feature
0	0	0	<a href="#">FMLALB (indexed)</a>	-
0	0	1	<a href="#">FMLALT (indexed)</a>	-
0	1	0	<a href="#">FMLS LB (indexed)</a>	-
0	1	1	<a href="#">FMLS LT (indexed)</a>	-
1	0	0	<a href="#">BFMLALB (indexed)</a>	FEAT_BF16
1	0	1	<a href="#">BFMLALT (indexed)</a>	FEAT_BF16
1	1	0	<a href="#">BFMLS LB (indexed)</a>	FEAT_SVE2p1
1	1	1	<a href="#">BFMLS LT (indexed)</a>	FEAT_SVE2p1

## SVE Floating Point Widening Multiply-Add

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01100100								op0		1						10	op1	00													

Decode fields		Instruction details
op0	op1	
0	0	<a href="#">SVE BFloat16 floating-point dot product</a>
0	1	UNALLOCATED
1		<a href="#">SVE floating-point multiply-add long</a>

### SVE BFloat16 floating-point dot product

These instructions are under [SVE Floating Point Widening Multiply-Add](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	0	op	1	Zm					1	0	0	0	0	o2	Zn					Zda				

Decode fields		Instruction Details	Feature
op	o2		
	1	UNALLOCATED	-
0	0	<a href="#">FDOT (vectors)</a>	FEAT_SVE2p1
1	0	<a href="#">BFDOT (vectors)</a>	FEAT_BF16

### SVE floating-point multiply-add long

These instructions are under [SVE Floating Point Widening Multiply-Add](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	o2	1	Zm					1	0	op	0	0	T	Zn					Zda				

Decode fields			Instruction Details	Feature
o2	op	T		
0	0	0	<a href="#">FMLALB (vectors)</a>	-
0	0	1	<a href="#">FMLALT (vectors)</a>	-
0	1	0	<a href="#">FMLS LB (vectors)</a>	-
0	1	1	<a href="#">FMLS LT (vectors)</a>	-
1	0	0	<a href="#">BFMLALB (vectors)</a>	FEAT_BF16
1	0	1	<a href="#">BFMLALT (vectors)</a>	FEAT_BF16
1	1	0	<a href="#">BFMLS LB (vectors)</a>	FEAT_SVE2p1
1	1	1	<a href="#">BFMLS LT (vectors)</a>	FEAT_SVE2p1

### SVE floating point matrix multiply accumulate

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	1	0	0	1	0	0	opc	1	Zm						1	1	1	0	0	1	Zn						Zda				

Decode fields		Instruction Details	Feature
opc			
00		UNALLOCATED	-
01		<a href="#">BFMMLA</a>	FEAT_BF16
10		<a href="#">FMMLA</a> — <a href="#">32-bit element</a>	FEAT_F32MM

Decode fields	Instruction Details	Feature
<b>opc</b>		
11	<a href="#">FMMLA — 64-bit element</a>	FEAT_F64MM

## SVE floating-point compare vectors

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	Zm				op	1	o2	Pg			Zn				o3	Pd						

Decode fields	Instruction Details		
<b>op</b>	<b>o2</b>	<b>o3</b>	
0	0	0	<a href="#">FCM&lt;cc&gt; (vectors) — FCMGE</a>
0	0	1	<a href="#">FCM&lt;cc&gt; (vectors) — FCMGT</a>
0	1	0	<a href="#">FCM&lt;cc&gt; (vectors) — FCMEQ</a>
0	1	1	<a href="#">FCM&lt;cc&gt; (vectors) — FCMNE</a>
1	0	0	<a href="#">FCM&lt;cc&gt; (vectors) — FCMUO</a>
1	0	1	<a href="#">FAC&lt;cc&gt; — FACGE</a>
1	1	0	UNALLOCATED
1	1	1	<a href="#">FAC&lt;cc&gt; — FACGT</a>

## SVE floating-point arithmetic (unpredicated)

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	Zm			0	0	0	opc			Zn			Zd									

Decode fields	Instruction Details		Feature
<b>size</b>	<b>opc</b>		
	011	<a href="#">FTSMUL</a>	-
	10x	UNALLOCATED	-
	110	<a href="#">FRECPS</a>	-
	111	<a href="#">FRSORTS</a>	-
!= 00	000	<a href="#">FADD (vectors, unpredicated)</a>	-
!= 00	001	<a href="#">FSUB (vectors, unpredicated)</a>	-
!= 00	010	<a href="#">FMUL (vectors, unpredicated)</a>	-
00	000	<a href="#">BFADD (unpredicated)</a>	FEAT_SVE_B16B16
00	001	<a href="#">BFSUB (unpredicated)</a>	FEAT_SVE_B16B16
00	010	<a href="#">BFMUL (vectors, unpredicated)</a>	FEAT_SVE_B16B16

## SVE Floating Point Arithmetic - Predicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										0	op0						100		op1		op2										

Decode fields			Instruction details
op0	op1	op2	
0x			<a href="#">SVE floating-point arithmetic (predicated)</a>
10	000		<a href="#">FTMAD</a>
10	!= 000		UNALLOCATED



11		0000	<a href="#">SVE floating-point arithmetic with immediate (predicated)</a>
11		!= 0000	UNALLOCATED

**SVE floating-point arithmetic (predicated)**

These instructions are under [SVE Floating Point Arithmetic - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0		opc	1	0	0		Pg							Zm					Zdn		

Decode fields size	opc	Instruction Details	Feature
	0011	<a href="#">FSUBR (vectors)</a>	-
	1000	<a href="#">FABD</a>	-
	1001	<a href="#">FSCALE</a>	-
	1010	<a href="#">FMULX</a>	-
	1011	UNALLOCATED	-
	1100	<a href="#">FDIVR</a>	-
	1101	<a href="#">FDIV</a>	-
	111x	UNALLOCATED	-
!= 00	0000	<a href="#">FADD (vectors, predicated)</a>	-
!= 00	0001	<a href="#">FSUB (vectors, predicated)</a>	-
!= 00	0010	<a href="#">FMUL (vectors, predicated)</a>	-
!= 00	0100	<a href="#">FMAXNM (vectors)</a>	-
!= 00	0101	<a href="#">FMINNM (vectors)</a>	-
!= 00	0110	<a href="#">FMAX (vectors)</a>	-
!= 00	0111	<a href="#">FMIN (vectors)</a>	-
00	0000	<a href="#">BFADD (predicated)</a>	FEAT_SVE_B16B16
00	0001	<a href="#">BFSUB (predicated)</a>	FEAT_SVE_B16B16
00	0010	<a href="#">BFMUL (vectors, predicated)</a>	FEAT_SVE_B16B16
00	0100	<a href="#">BFMAXNM</a>	FEAT_SVE_B16B16
00	0101	<a href="#">BFMINNM</a>	FEAT_SVE_B16B16
00	0110	<a href="#">BFMAX</a>	FEAT_SVE_B16B16
00	0111	<a href="#">BFMIN</a>	FEAT_SVE_B16B16

**SVE floating-point arithmetic with immediate (predicated)**

These instructions are under [SVE Floating Point Arithmetic - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	1	1		opc	1	0	0		Pg					0	0	0	0	i1			Zdn	

Decode fields opc	Instruction Details
000	<a href="#">FADD (immediate)</a>
001	<a href="#">FSUB (immediate)</a>
010	<a href="#">FMUL (immediate)</a>
011	<a href="#">FSUBR (immediate)</a>
100	<a href="#">FMAXNM (immediate)</a>
101	<a href="#">FMINNM (immediate)</a>
110	<a href="#">FMAX (immediate)</a>
111	<a href="#">FMIN (immediate)</a>

## SVE Floating Point Unary Operations - Predicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01100101										0	op0						101														

Decode fields op0	Instruction details
00x	<a href="#">SVE floating-point round to integral value</a>
010	<a href="#">SVE floating-point convert precision</a>
011	<a href="#">SVE floating-point unary operations</a>
10x	<a href="#">SVE integer convert to floating-point</a>
11x	<a href="#">SVE floating-point convert to integer</a>

### SVE floating-point round to integral value

These instructions are under [SVE Floating Point Unary Operations - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	0	0	opc		1	0	1	Pg		Zn				Zd							

Decode fields opc	Instruction Details
000	<a href="#">FRINT&lt;r&gt; — nearest with ties to even</a>
001	<a href="#">FRINT&lt;r&gt; — toward plus infinity</a>
010	<a href="#">FRINT&lt;r&gt; — toward minus infinity</a>
011	<a href="#">FRINT&lt;r&gt; — toward zero</a>
100	<a href="#">FRINT&lt;r&gt; — nearest with ties to away</a>
101	UNALLOCATED
110	<a href="#">FRINT&lt;r&gt; — current mode signalling inexact</a>
111	<a href="#">FRINT&lt;r&gt; — current mode</a>

### SVE floating-point convert precision

These instructions are under [SVE Floating Point Unary Operations - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	opc		0	0	1	0	opc2		1	0	1	Pg		Zn				Zd						

Decode fields opc	Decode fields opc2	Instruction Details	Feature
x0	11	UNALLOCATED	-
00	0x	UNALLOCATED	-
00	10	<a href="#">FCVTX</a>	-
01		UNALLOCATED	-
10	00	<a href="#">FCVT — single-precision to half-precision</a>	-
10	01	<a href="#">FCVT — half-precision to single-precision</a>	-
10	10	<a href="#">BFCVT</a>	FEAT_BF16
11	00	<a href="#">FCVT — double-precision to half-precision</a>	-
11	01	<a href="#">FCVT — half-precision to double-precision</a>	-
11	10	<a href="#">FCVT — double-precision to single-precision</a>	-
11	11	<a href="#">FCVT — single-precision to double-precision</a>	-

**SVE floating-point unary operations**

These instructions are under [SVE Floating Point Unary Operations - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	0	1	1	opc		1	0	1	Pg		Zn				Zd						

Decode fields opc	Instruction Details
00	<a href="#">FRECPX</a>
01	<a href="#">FSQRT</a>
1x	UNALLOCATED

**SVE integer convert to floating-point**

These instructions are under [SVE Floating Point Unary Operations - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	opc		0	1	0	opc2		U	1	0	1	Pg		Zn				Zd						

Decode fields opc	opc2	U	Instruction Details
00			UNALLOCATED
01	00		UNALLOCATED
01	01	0	<a href="#">SCVTF — 16-bit to half-precision</a>
01	01	1	<a href="#">UCVTF — 16-bit to half-precision</a>
01	10	0	<a href="#">SCVTF — 32-bit to half-precision</a>
01	10	1	<a href="#">UCVTF — 32-bit to half-precision</a>
01	11	0	<a href="#">SCVTF — 64-bit to half-precision</a>
01	11	1	<a href="#">UCVTF — 64-bit to half-precision</a>
10	0x		UNALLOCATED
10	10	0	<a href="#">SCVTF — 32-bit to single-precision</a>
10	10	1	<a href="#">UCVTF — 32-bit to single-precision</a>
10	11		UNALLOCATED
11	00	0	<a href="#">SCVTF — 32-bit to double-precision</a>
11	00	1	<a href="#">UCVTF — 32-bit to double-precision</a>
11	01		UNALLOCATED
11	10	0	<a href="#">SCVTF — 64-bit to single-precision</a>
11	10	1	<a href="#">UCVTF — 64-bit to single-precision</a>
11	11	0	<a href="#">SCVTF — 64-bit to double-precision</a>
11	11	1	<a href="#">UCVTF — 64-bit to double-precision</a>

**SVE floating-point convert to integer**

These instructions are under [SVE Floating Point Unary Operations - Predicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	opc		0	1	1	opc2		U	1	0	1	Pg		Zn				Zd						

Decode fields opc	opc2	U	Instruction Details
00		0	<a href="#">FLOGB</a>
00		1	UNALLOCATED
01	00		UNALLOCATED
01	01	0	<a href="#">FCVTZS — half-precision to 16-bit</a>

Decode fields			Instruction Details
opc	opc2	U	
01	01	1	<a href="#">FCVTZU — half-precision to 16-bit</a>
01	10	0	<a href="#">FCVTZS — half-precision to 32-bit</a>
01	10	1	<a href="#">FCVTZU — half-precision to 32-bit</a>
01	11	0	<a href="#">FCVTZS — half-precision to 64-bit</a>
01	11	1	<a href="#">FCVTZU — half-precision to 64-bit</a>
10	0x		UNALLOCATED
10	10	0	<a href="#">FCVTZS — single-precision to 32-bit</a>
10	10	1	<a href="#">FCVTZU — single-precision to 32-bit</a>
10	11		UNALLOCATED
11	00	0	<a href="#">FCVTZS — double-precision to 32-bit</a>
11	00	1	<a href="#">FCVTZU — double-precision to 32-bit</a>
11	01		UNALLOCATED
11	10	0	<a href="#">FCVTZS — single-precision to 64-bit</a>
11	10	1	<a href="#">FCVTZU — single-precision to 64-bit</a>
11	11	0	<a href="#">FCVTZS — double-precision to 64-bit</a>
11	11	1	<a href="#">FCVTZU — double-precision to 64-bit</a>

### SVE floating-point recursive reduction

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	0	opc	0	0	1	Pg	Zn				Vd										

Decode fields		Instruction Details
opc		
000		<a href="#">FADDV</a>
001		UNALLOCATED
01x		UNALLOCATED
100		<a href="#">FMAXNMV</a>
101		<a href="#">FMINNMV</a>
110		<a href="#">FMAXV</a>
111		<a href="#">FMINV</a>

### SVE Floating Point Unary Operations - Unpredicated

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01100101										001		op0				0011				op1											

Decode fields		Instruction details
op0	op1	
0x		UNALLOCATED
10		UNALLOCATED
11	00	<a href="#">SVE floating-point reciprocal estimate (unpredicated)</a>
11	!= 00	UNALLOCATED

### SVE floating-point reciprocal estimate (unpredicated)

These instructions are under [SVE Floating Point Unary Operations - Unpredicated](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	0	1	1	1	op	0	0	1	1	0	0	Zn				Zd						

### Decode fields

op	Instruction Details
----	---------------------

0	<a href="#">FRECPE</a>
1	<a href="#">FRSQRT</a>

## SVE Floating Point Compare - with Zero

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01100101										010	op0			001																	

### Decode fields

op0	Instruction details
-----	---------------------

0	<a href="#">SVE floating-point compare with zero</a>
1	UNALLOCATED

## SVE floating-point compare with zero

These instructions are under [SVE Floating Point Compare - with Zero](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	1	0	0	eq	lt	0	0	1	Pg				Zn				ne	Pd				

### Decode fields

eq	lt	ne	Instruction Details
----	----	----	---------------------

0	0	0	<a href="#">FCM&lt;cc&gt; (zero) — FCMGE</a>
0	0	1	<a href="#">FCM&lt;cc&gt; (zero) — FCMGT</a>
0	1	0	<a href="#">FCM&lt;cc&gt; (zero) — FCMLT</a>
0	1	1	<a href="#">FCM&lt;cc&gt; (zero) — FCMLE</a>
1		1	UNALLOCATED
1	0	0	<a href="#">FCM&lt;cc&gt; (zero) — FCMEQ</a>
1	1	0	<a href="#">FCM&lt;cc&gt; (zero) — FCMNE</a>

## SVE Floating Point Accumulating Reduction

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01100101										011	op0					001															

### Decode fields

op0	Instruction details
-----	---------------------

0	<a href="#">SVE floating-point serial reduction (predicated)</a>
1	UNALLOCATED

## SVE floating-point serial reduction (predicated)

These instructions are under [SVE Floating Point Accumulating Reduction](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	0	1	1	0	opc	0	0	1	Pg				Zm				Vdn						

Decode fields opc	Instruction Details
00	<a href="#">FADDA</a>
01	UNALLOCATED
1x	UNALLOCATED

## SVE Floating Point Multiply-Add

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

Decode fields op0	Instruction details
0	<a href="#">SVE floating-point multiply-accumulate writing addend</a>
1	<a href="#">SVE floating-point multiply-accumulate writing multiplicand</a>

### SVE floating-point multiply-accumulate writing addend

These instructions are under [SVE Floating Point Multiply-Add](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

Decode fields size	opc	Instruction Details	Feature
	10	<a href="#">FNMLA</a>	-
	11	<a href="#">FNMLS</a>	-
!= 00	00	<a href="#">FMLA (vectors)</a>	-
!= 00	01	<a href="#">FMLS (vectors)</a>	-
00	00	<a href="#">BFMLA (vectors)</a>	FEAT_SVE_B16B16
00	01	<a href="#">BFMLS (vectors)</a>	FEAT_SVE_B16B16

### SVE floating-point multiply-accumulate writing multiplicand

These instructions are under [SVE Floating Point Multiply-Add](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

Decode fields opc	Instruction Details
00	<a href="#">FMAD</a>
01	<a href="#">FMSB</a>
10	<a href="#">FNMAAD</a>
11	<a href="#">FNMSB</a>

## SVE Memory - 32-bit Gather and Unsized Contiguous

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Decode fields				Instruction details
op0	op1	op2	op3	
00	x1	0xx	0	<a href="#">SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)</a>
00	x1	0xx	1	UNALLOCATED
01	x1	0xx		<a href="#">SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets)</a>
10	x1	0xx		<a href="#">SVE 32-bit gather load words (scalar plus 32-bit scaled offsets)</a>
11	0x	000	0	<a href="#">LDR (predicate)</a>
11	0x	000	1	UNALLOCATED
11	0x	010		<a href="#">LDR (vector)</a>
11	0x	0x1		UNALLOCATED
11	1x	0xx	0	<a href="#">SVE contiguous prefetch (scalar plus immediate)</a>
11	1x	0xx	1	UNALLOCATED
!= 11	x0	0xx		<a href="#">SVE 32-bit gather load (scalar plus 32-bit unscaled offsets)</a>
	00	10x		<a href="#">SVE2 32-bit gather non-temporal load (vector plus scalar)</a>
	00	110	0	<a href="#">SVE contiguous prefetch (scalar plus scalar)</a>
	00	111	0	<a href="#">SVE 32-bit gather prefetch (vector plus immediate)</a>
	00	11x	1	UNALLOCATED
	01	1xx		<a href="#">SVE 32-bit gather load (vector plus immediate)</a>
	1x	1xx		<a href="#">SVE load and broadcast element</a>

**SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)**

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	0	xs	1			Zm		0	msz		Pg						Rn		0			prfop		

Decode fields		Instruction Details
msz		
00		<a href="#">PRFB (scalar plus vector)</a>
01		<a href="#">PRFH (scalar plus vector)</a>
10		<a href="#">PRFW (scalar plus vector)</a>
11		<a href="#">PRFD (scalar plus vector)</a>

**SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets)**

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	1	xs	1			Zm		0	U	ff		Pg						Rn				Zt		

Decode fields		Instruction Details
U	ff	
0	0	<a href="#">LD1SH (scalar plus vector)</a>
0	1	<a href="#">LDFF1SH (scalar plus vector)</a>
1	0	<a href="#">LD1H (scalar plus vector)</a>
1	1	<a href="#">LDFF1H (scalar plus vector)</a>

**SVE 32-bit gather load words (scalar plus 32-bit scaled offsets)**

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1	0	xs	1			Zm		0	U	ff		Pg						Rn				Zt		

Decode fields		Instruction Details
U	ff	
0		UNALLOCATED
1	0	<a href="#">LD1W (scalar plus vector)</a>
1	1	<a href="#">LDFF1W (scalar plus vector)</a>

**SVE contiguous prefetch (scalar plus immediate)**

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1	1	1	imm6						0	msz	Pg				Rn				0	prfop				

Decode fields		Instruction Details
msz		
00		<a href="#">PRFB (scalar plus immediate)</a>
01		<a href="#">PRFH (scalar plus immediate)</a>
10		<a href="#">PRFW (scalar plus immediate)</a>
11		<a href="#">PRFD (scalar plus immediate)</a>

**SVE 32-bit gather load (scalar plus 32-bit unscaled offsets)**

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	!= 11	xs	0	Zm						0	U	ff	Pg				Rn				Zt				
opc																															

The following constraints also apply to this encoding: `opc != 11 && opc != 11`

Decode fields			Instruction Details
opc	U	ff	
00	0	0	<a href="#">LD1SB (scalar plus vector)</a>
00	0	1	<a href="#">LDFF1SB (scalar plus vector)</a>
00	1	0	<a href="#">LD1B (scalar plus vector)</a>
00	1	1	<a href="#">LDFF1B (scalar plus vector)</a>
01	0	0	<a href="#">LD1SH (scalar plus vector)</a>
01	0	1	<a href="#">LDFF1SH (scalar plus vector)</a>
01	1	0	<a href="#">LD1H (scalar plus vector)</a>
01	1	1	<a href="#">LDFF1H (scalar plus vector)</a>
10	0		UNALLOCATED
10	1	0	<a href="#">LD1W (scalar plus vector)</a>
10	1	1	<a href="#">LDFF1W (scalar plus vector)</a>

**SVE2 32-bit gather non-temporal load (vector plus scalar)**

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	msz	0	0	Rm						1	0	U	Pg				Zn				Zt				

Decode fields		Instruction Details
msz	U	
00	0	<a href="#">LDNT1SB</a>



Decode fields		Instruction Details
msz	U	
00	1	<a href="#">LDNT1B (vector plus scalar)</a>
01	0	<a href="#">LDNT1SH</a>
01	1	<a href="#">LDNT1H (vector plus scalar)</a>
10	0	UNALLOCATED
10	1	<a href="#">LDNT1W (vector plus scalar)</a>
11		UNALLOCATED

**SVE contiguous prefetch (scalar plus scalar)**

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	msz	0	0		Rm					1	1	0		Pg							0			prfop	

Decode fields		Instruction Details
msz		
00		<a href="#">PRFB (scalar plus scalar)</a>
01		<a href="#">PRFH (scalar plus scalar)</a>
10		<a href="#">PRFW (scalar plus scalar)</a>
11		<a href="#">PRFD (scalar plus scalar)</a>

**SVE 32-bit gather prefetch (vector plus immediate)**

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	msz	0	0		imm5					1	1	1		Pg							0			prfop	

Decode fields		Instruction Details
msz		
00		<a href="#">PRFB (vector plus immediate)</a>
01		<a href="#">PRFH (vector plus immediate)</a>
10		<a href="#">PRFW (vector plus immediate)</a>
11		<a href="#">PRFD (vector plus immediate)</a>

**SVE 32-bit gather load (vector plus immediate)**

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	msz	0	1	imm5						1	U	ff	Pg			Zn				Zt					

Decode fields			Instruction Details
msz	U	ff	
00	0	0	<a href="#">LD1SB (vector plus immediate)</a>
00	0	1	<a href="#">LDFF1SB (vector plus immediate)</a>
00	1	0	<a href="#">LD1B (vector plus immediate)</a>
00	1	1	<a href="#">LDFF1B (vector plus immediate)</a>
01	0	0	<a href="#">LD1SH (vector plus immediate)</a>
01	0	1	<a href="#">LDFF1SH (vector plus immediate)</a>
01	1	0	<a href="#">LD1H (vector plus immediate)</a>
01	1	1	<a href="#">LDFF1H (vector plus immediate)</a>

Decode fields			Instruction Details
msz	U	ff	
10	0		UNALLOCATED
10	1	0	<a href="#">LD1W (vector plus immediate)</a>
10	1	1	<a href="#">LDFF1W (vector plus immediate)</a>
11			UNALLOCATED

### SVE load and broadcast element

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	dtypeh		1	imm6						1	dtypel		Pg		Rn				Zt						

Decode fields		Instruction Details
dtypeh	dtypel	
00	00	<a href="#">LD1RB — 8-bit element</a>
00	01	<a href="#">LD1RB — 16-bit element</a>
00	10	<a href="#">LD1RB — 32-bit element</a>
00	11	<a href="#">LD1RB — 64-bit element</a>
01	00	<a href="#">LD1RSW</a>
01	01	<a href="#">LD1RH — 16-bit element</a>
01	10	<a href="#">LD1RH — 32-bit element</a>
01	11	<a href="#">LD1RH — 64-bit element</a>
10	00	<a href="#">LD1RSH — 64-bit element</a>
10	01	<a href="#">LD1RSH — 32-bit element</a>
10	10	<a href="#">LD1RW — 32-bit element</a>
10	11	<a href="#">LD1RW — 64-bit element</a>
11	00	<a href="#">LD1RSB — 64-bit element</a>
11	01	<a href="#">LD1RSB — 32-bit element</a>
11	10	<a href="#">LD1RSB — 16-bit element</a>
11	11	<a href="#">LD1RD</a>

### SVE Memory - Contiguous Load

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1010010								op0		op1					op2																

Decode fields			Instruction details
op0	op1	op2	
00	0	111	<a href="#">SVE contiguous non-temporal load (scalar plus immediate)</a>
00	1	001	<a href="#">SVE contiguous load (quadwords, scalar plus immediate)</a>
00	1	111	<a href="#">SVE load multiple structures (quadwords, scalar plus immediate)</a>
00		100	<a href="#">SVE contiguous load (quadwords, scalar plus scalar)</a>
00		110	<a href="#">SVE contiguous non-temporal load (scalar plus scalar)</a>
01		100	<a href="#">SVE load multiple structures (quadwords, scalar plus scalar)</a>
1x		100	UNALLOCATED
!= 00	0	111	<a href="#">SVE load multiple structures (scalar plus immediate)</a>
!= 00	1	001	UNALLOCATED
!= 00	1	111	UNALLOCATED

!= 00		110	<a href="#">SVE load multiple structures (scalar plus scalar)</a>
	0	001	<a href="#">SVE load and broadcast quadword (scalar plus immediate)</a>
	0	101	<a href="#">SVE contiguous load (scalar plus immediate)</a>
	1	101	<a href="#">SVE contiguous non-fault load (scalar plus immediate)</a>
		000	<a href="#">SVE load and broadcast quadword (scalar plus scalar)</a>
		010	<a href="#">SVE contiguous load (scalar plus scalar)</a>
		011	<a href="#">SVE contiguous first-fault load (scalar plus scalar)</a>

**SVE contiguous non-temporal load (scalar plus immediate)**

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	msz		0	0	0	imm4			1	1	1	Pg			Rn			Zt							

**Decode fields**  
**msz****Instruction Details**

00	<a href="#">LDNT1B (scalar plus immediate, single register)</a>
01	<a href="#">LDNT1H (scalar plus immediate, single register)</a>
10	<a href="#">LDNT1W (scalar plus immediate, single register)</a>
11	<a href="#">LDNT1D (scalar plus immediate, single register)</a>

**SVE contiguous load (quadwords, scalar plus immediate)**

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	dtype		0	0	1	imm4			0	0	1	Pg			Rn			Zt							

**Decode fields**  
**dtype****Instruction Details****Feature**

0x	UNALLOCATED	-
10	<a href="#">LD1W (scalar plus immediate, single register)</a>	FEAT_SVE2p1
11	<a href="#">LD1D (scalar plus immediate, single register)</a>	FEAT_SVE2p1

**SVE load multiple structures (quadwords, scalar plus immediate)**

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	num		0	0	1	imm4			1	1	1	Pg			Rn			Zt							

**Decode fields**  
**num****Instruction Details****Feature**

00	UNALLOCATED	-
01	<a href="#">LD2Q (scalar plus immediate)</a>	FEAT_SVE2p1
10	<a href="#">LD3Q (scalar plus immediate)</a>	FEAT_SVE2p1
11	<a href="#">LD4Q (scalar plus immediate)</a>	FEAT_SVE2p1

**SVE contiguous load (quadwords, scalar plus scalar)**

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	dtype		0	0	Rm			1	0	0	Pg			Rn			Zt								

Decode fields dtype	Instruction Details	Feature
0x	UNALLOCATED	-
10	<a href="#">LD1W (scalar plus scalar, single register)</a>	FEAT_SVE2p1
11	<a href="#">LD1D (scalar plus scalar, single register)</a>	FEAT_SVE2p1

**SVE contiguous non-temporal load (scalar plus scalar)**

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	msz	0	0		Rm					1	1	0		Pg			Rn				Zt				

Decode fields msz	Instruction Details
00	<a href="#">LDNT1B (scalar plus scalar, single register)</a>
01	<a href="#">LDNT1H (scalar plus scalar, single register)</a>
10	<a href="#">LDNT1W (scalar plus scalar, single register)</a>
11	<a href="#">LDNT1D (scalar plus scalar, single register)</a>

**SVE load multiple structures (quadwords, scalar plus scalar)**

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	num	0	1	Rm						1	0	0	Pg			Rn				Zt					

Decode fields num	Instruction Details	Feature
00	UNALLOCATED	-
01	<a href="#">LD2Q (scalar plus scalar)</a>	FEAT_SVE2p1
10	<a href="#">LD3Q (scalar plus scalar)</a>	FEAT_SVE2p1
11	<a href="#">LD4Q (scalar plus scalar)</a>	FEAT_SVE2p1

**SVE load multiple structures (scalar plus immediate)**

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1 0 1 0 0 1 0						msz		!= 00		0		imm4			1 1 1			Pg			Rn					Zt					
opc																															

The following constraints also apply to this encoding: `opc != 00 && opc != 00`

Decode fields msz opc	Instruction Details
00 01	<a href="#">LD2B (scalar plus immediate)</a>
00 10	<a href="#">LD3B (scalar plus immediate)</a>
00 11	<a href="#">LD4B (scalar plus immediate)</a>
01 01	<a href="#">LD2H (scalar plus immediate)</a>
01 10	<a href="#">LD3H (scalar plus immediate)</a>
01 11	<a href="#">LD4H (scalar plus immediate)</a>
10 01	<a href="#">LD2W (scalar plus immediate)</a>
10 10	<a href="#">LD3W (scalar plus immediate)</a>

Decode fields		Instruction Details
msz	opc	
10	11	<a href="#">LD4W (scalar plus immediate)</a>
11	01	<a href="#">LD2D (scalar plus immediate)</a>
11	10	<a href="#">LD3D (scalar plus immediate)</a>
11	11	<a href="#">LD4D (scalar plus immediate)</a>

**SVE load multiple structures (scalar plus scalar)**

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	msz		!= 00		Rm				1 1 0			Pg			Rn				Zt						
opc																															

The following constraints also apply to this encoding: opc != 00 && opc != 00

Decode fields		Instruction Details
msz	opc	
00	01	<a href="#">LD2B (scalar plus scalar)</a>
00	10	<a href="#">LD3B (scalar plus scalar)</a>
00	11	<a href="#">LD4B (scalar plus scalar)</a>
01	01	<a href="#">LD2H (scalar plus scalar)</a>
01	10	<a href="#">LD3H (scalar plus scalar)</a>
01	11	<a href="#">LD4H (scalar plus scalar)</a>
10	01	<a href="#">LD2W (scalar plus scalar)</a>
10	10	<a href="#">LD3W (scalar plus scalar)</a>
10	11	<a href="#">LD4W (scalar plus scalar)</a>
11	01	<a href="#">LD2D (scalar plus scalar)</a>
11	10	<a href="#">LD3D (scalar plus scalar)</a>
11	11	<a href="#">LD4D (scalar plus scalar)</a>

**SVE load and broadcast quadword (scalar plus immediate)**

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	msz		ssz		0	imm4				0	0	1	Pg			Rn				Zt					

Decode fields		Instruction Details	Feature
msz	ssz		
	1x	UNALLOCATED	-
00	00	<a href="#">LD1RQB (scalar plus immediate)</a>	-
00	01	<a href="#">LD1ROB (scalar plus immediate)</a>	FEAT_F64MM
01	00	<a href="#">LD1RQH (scalar plus immediate)</a>	-
01	01	<a href="#">LD1ROH (scalar plus immediate)</a>	FEAT_F64MM
10	00	<a href="#">LD1RQW (scalar plus immediate)</a>	-
10	01	<a href="#">LD1ROW (scalar plus immediate)</a>	FEAT_F64MM
11	00	<a href="#">LD1RQD (scalar plus immediate)</a>	-
11	01	<a href="#">LD1ROD (scalar plus immediate)</a>	FEAT_F64MM

**SVE contiguous load (scalar plus immediate)**

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	dtype				0	imm4				1	0	1	Pg			Rn				Zt					

Decode fields dtype	Instruction Details
0000	<a href="#">LD1B (scalar plus immediate, single register)</a> — 8-bit element
0001	<a href="#">LD1B (scalar plus immediate, single register)</a> — 16-bit element
0010	<a href="#">LD1B (scalar plus immediate, single register)</a> — 32-bit element
0011	<a href="#">LD1B (scalar plus immediate, single register)</a> — 64-bit element
0100	<a href="#">LD1SW (scalar plus immediate)</a>
0101	<a href="#">LD1H (scalar plus immediate, single register)</a> — 16-bit element
0110	<a href="#">LD1H (scalar plus immediate, single register)</a> — 32-bit element
0111	<a href="#">LD1H (scalar plus immediate, single register)</a> — 64-bit element
1000	<a href="#">LD1SH (scalar plus immediate)</a> — 64-bit element
1001	<a href="#">LD1SH (scalar plus immediate)</a> — 32-bit element
1010	<a href="#">LD1W (scalar plus immediate, single register)</a> — 32-bit element
1011	<a href="#">LD1W (scalar plus immediate, single register)</a> — 64-bit element
1100	<a href="#">LD1SB (scalar plus immediate)</a> — 64-bit element
1101	<a href="#">LD1SB (scalar plus immediate)</a> — 32-bit element
1110	<a href="#">LD1SB (scalar plus immediate)</a> — 16-bit element
1111	<a href="#">LD1D (scalar plus immediate, single register)</a>

**SVE contiguous non-fault load (scalar plus immediate)**

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	dtype				1	imm4				1	0	1	Pg			Rn				Zt					

Decode fields dtype	Instruction Details
0000	<a href="#">LDNF1B</a> — 8-bit element
0001	<a href="#">LDNF1B</a> — 16-bit element
0010	<a href="#">LDNF1B</a> — 32-bit element
0011	<a href="#">LDNF1B</a> — 64-bit element
0100	<a href="#">LDNF1SW</a>
0101	<a href="#">LDNF1H</a> — 16-bit element
0110	<a href="#">LDNF1H</a> — 32-bit element
0111	<a href="#">LDNF1H</a> — 64-bit element
1000	<a href="#">LDNF1SH</a> — 64-bit element
1001	<a href="#">LDNF1SH</a> — 32-bit element
1010	<a href="#">LDNF1W</a> — 32-bit element
1011	<a href="#">LDNF1W</a> — 64-bit element
1100	<a href="#">LDNF1SB</a> — 64-bit element
1101	<a href="#">LDNF1SB</a> — 32-bit element
1110	<a href="#">LDNF1SB</a> — 16-bit element
1111	<a href="#">LDNF1D</a>

**SVE load and broadcast quadword (scalar plus scalar)**

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	msz		ssz		Rm				0	0	0	Pg		Rn				Zt							

Decode fields		Instruction Details				Feature
msz	ssz					
	1x	UNALLOCATED				-
00	00	<a href="#">LD1RQB (scalar plus scalar)</a>				-
00	01	<a href="#">LD1ROB (scalar plus scalar)</a>				FEAT_F64MM
01	00	<a href="#">LD1RQH (scalar plus scalar)</a>				-
01	01	<a href="#">LD1ROH (scalar plus scalar)</a>				FEAT_F64MM
10	00	<a href="#">LD1RQW (scalar plus scalar)</a>				-
10	01	<a href="#">LD1ROW (scalar plus scalar)</a>				FEAT_F64MM
11	00	<a href="#">LD1RQD (scalar plus scalar)</a>				-
11	01	<a href="#">LD1ROD (scalar plus scalar)</a>				FEAT_F64MM

**SVE contiguous load (scalar plus scalar)**

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	dtype		Rm				0	1	0	Pg		Rn				Zt									

Decode fields		Instruction Details			
dtype					
0000		<a href="#">LD1B (scalar plus scalar, single register) — 8-bit element</a>			
0001		<a href="#">LD1B (scalar plus scalar, single register) — 16-bit element</a>			
0010		<a href="#">LD1B (scalar plus scalar, single register) — 32-bit element</a>			
0011		<a href="#">LD1B (scalar plus scalar, single register) — 64-bit element</a>			
0100		<a href="#">LD1SW (scalar plus scalar)</a>			
0101		<a href="#">LD1H (scalar plus scalar, single register) — 16-bit element</a>			
0110		<a href="#">LD1H (scalar plus scalar, single register) — 32-bit element</a>			
0111		<a href="#">LD1H (scalar plus scalar, single register) — 64-bit element</a>			
1000		<a href="#">LD1SH (scalar plus scalar) — 64-bit element</a>			
1001		<a href="#">LD1SH (scalar plus scalar) — 32-bit element</a>			
1010		<a href="#">LD1W (scalar plus scalar, single register) — 32-bit element</a>			
1011		<a href="#">LD1W (scalar plus scalar, single register) — 64-bit element</a>			
1100		<a href="#">LD1SB (scalar plus scalar) — 64-bit element</a>			
1101		<a href="#">LD1SB (scalar plus scalar) — 32-bit element</a>			
1110		<a href="#">LD1SB (scalar plus scalar) — 16-bit element</a>			
1111		<a href="#">LD1D (scalar plus scalar, single register)</a>			

**SVE contiguous first-fault load (scalar plus scalar)**

These instructions are under [SVE Memory - Contiguous Load](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	dtype		Rm				0	1	1	Pg		Rn				Zt									

Decode fields		Instruction Details			
dtype					
0000		<a href="#">LDFF1B (scalar plus scalar) — 8-bit element</a>			

Decode fields dtype	Instruction Details
0001	<a href="#">LDFF1B (scalar plus scalar) — 16-bit element</a>
0010	<a href="#">LDFF1B (scalar plus scalar) — 32-bit element</a>
0011	<a href="#">LDFF1B (scalar plus scalar) — 64-bit element</a>
0100	<a href="#">LDFF1SW (scalar plus scalar)</a>
0101	<a href="#">LDFF1H (scalar plus scalar) — 16-bit element</a>
0110	<a href="#">LDFF1H (scalar plus scalar) — 32-bit element</a>
0111	<a href="#">LDFF1H (scalar plus scalar) — 64-bit element</a>
1000	<a href="#">LDFF1SH (scalar plus scalar) — 64-bit element</a>
1001	<a href="#">LDFF1SH (scalar plus scalar) — 32-bit element</a>
1010	<a href="#">LDFF1W (scalar plus scalar) — 32-bit element</a>
1011	<a href="#">LDFF1W (scalar plus scalar) — 64-bit element</a>
1100	<a href="#">LDFF1SB (scalar plus scalar) — 64-bit element</a>
1101	<a href="#">LDFF1SB (scalar plus scalar) — 32-bit element</a>
1110	<a href="#">LDFF1SB (scalar plus scalar) — 16-bit element</a>
1111	<a href="#">LDFF1D (scalar plus scalar)</a>

## SVE Memory - 64-bit Gather

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1100010								op0		op1						op2								op3							

Decode fields				Instruction details												Feature	
op0	op1	op2	op3														
00	00	101		<a href="#">LD1Q</a>												FEAT_SVE2p1	
00	01	0xx	1	UNALLOCATED												-	
00	11	1xx	0	<a href="#">SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)</a>												-	
00	11		1	UNALLOCATED												-	
00	x1	0xx	0	<a href="#">SVE 64-bit gather prefetch (scalar plus unpacked 32-bit scaled offsets)</a>												-	
!= 00	00	101		UNALLOCATED												-	
!= 00	11	1xx		<a href="#">SVE 64-bit gather load (scalar plus 64-bit scaled offsets)</a>												-	
!= 00	x1	0xx		<a href="#">SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets)</a>												-	
	00	111	0	<a href="#">SVE 64-bit gather prefetch (vector plus immediate)</a>												-	
	00	111	1	UNALLOCATED												-	
	00	1x0		<a href="#">SVE2 64-bit gather non-temporal load (vector plus scalar)</a>												-	
	01	1xx		<a href="#">SVE 64-bit gather load (vector plus immediate)</a>												-	
	10	1xx		<a href="#">SVE 64-bit gather load (scalar plus 64-bit unscaled offsets)</a>												-	
	x0	0xx		<a href="#">SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets)</a>												-	

## SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)

These instructions are under [SVE Memory - 64-bit Gather](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	1	1	Zm			1	msz	Pg		Rn			0	prfop									



Decode fields msz	Instruction Details
00	<a href="#">PRFB (scalar plus vector)</a>
01	<a href="#">PRFH (scalar plus vector)</a>
10	<a href="#">PRFW (scalar plus vector)</a>
11	<a href="#">PRFD (scalar plus vector)</a>

**SVE 64-bit gather prefetch (scalar plus unpacked 32-bit scaled offsets)**

These instructions are under [SVE Memory - 64-bit Gather](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	xs	1	Zm			0	msz		Pg		Rn			0	prfop								

Decode fields msz	Instruction Details
00	<a href="#">PRFB (scalar plus vector)</a>
01	<a href="#">PRFH (scalar plus vector)</a>
10	<a href="#">PRFW (scalar plus vector)</a>
11	<a href="#">PRFD (scalar plus vector)</a>

**SVE 64-bit gather load (scalar plus 64-bit scaled offsets)**

These instructions are under [SVE Memory - 64-bit Gather](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	!= 00	1	1	Zm				1	U	ff	Pg			Rn				Zt							
opc																															

The following constraints also apply to this encoding: `opc != 00 && opc != 00`

Decode fields opc	U	ff	Instruction Details
01	0	0	<a href="#">LD1SH (scalar plus vector)</a>
01	0	1	<a href="#">LDFF1SH (scalar plus vector)</a>
01	1	0	<a href="#">LD1H (scalar plus vector)</a>
01	1	1	<a href="#">LDFF1H (scalar plus vector)</a>
10	0	0	<a href="#">LD1SW (scalar plus vector)</a>
10	0	1	<a href="#">LDFF1SW (scalar plus vector)</a>
10	1	0	<a href="#">LD1W (scalar plus vector)</a>
10	1	1	<a href="#">LDFF1W (scalar plus vector)</a>
11	0		UNALLOCATED
11	1	0	<a href="#">LD1D (scalar plus vector)</a>
11	1	1	<a href="#">LDFF1D (scalar plus vector)</a>

**SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets)**

These instructions are under [SVE Memory - 64-bit Gather](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1 1 0 0 0 1 0							!= 00	xs	1	Zm						0	U	ff	Pg				Rn				Zt				
opc																															

The following constraints also apply to this encoding: `opc != 00 && opc != 00`

Decode fields			Instruction Details
opc	U	ff	
01	0	0	<a href="#">LD1SH (scalar plus vector)</a>
01	0	1	<a href="#">LDFF1SH (scalar plus vector)</a>
01	1	0	<a href="#">LD1H (scalar plus vector)</a>
01	1	1	<a href="#">LDFF1H (scalar plus vector)</a>
10	0	0	<a href="#">LD1SW (scalar plus vector)</a>
10	0	1	<a href="#">LDFF1SW (scalar plus vector)</a>
10	1	0	<a href="#">LD1W (scalar plus vector)</a>
10	1	1	<a href="#">LDFF1W (scalar plus vector)</a>
11	0		UNALLOCATED
11	1	0	<a href="#">LD1D (scalar plus vector)</a>
11	1	1	<a href="#">LDFF1D (scalar plus vector)</a>

**SVE 64-bit gather prefetch (vector plus immediate)**

These instructions are under [SVE Memory - 64-bit Gather](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	msz		0	0	imm5				1	1	1	Pg			Zn			0	prfop						

Decode fields		Instruction Details
msz		
00		<a href="#">PRFB (vector plus immediate)</a>
01		<a href="#">PRFH (vector plus immediate)</a>
10		<a href="#">PRFW (vector plus immediate)</a>
11		<a href="#">PRFD (vector plus immediate)</a>

**SVE2 64-bit gather non-temporal load (vector plus scalar)**

These instructions are under [SVE Memory - 64-bit Gather](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	1	0	msz		0	0	Rm				1	U	0	Pg			Zn			Zt								

Decode fields		Instruction Details
msz	U	
00	0	<a href="#">LDNT1SB</a>
00	1	<a href="#">LDNT1B (vector plus scalar)</a>
01	0	<a href="#">LDNT1SH</a>
01	1	<a href="#">LDNT1H (vector plus scalar)</a>
10	0	<a href="#">LDNT1SW</a>
10	1	<a href="#">LDNT1W (vector plus scalar)</a>
11	0	UNALLOCATED
11	1	<a href="#">LDNT1D (vector plus scalar)</a>

**SVE 64-bit gather load (vector plus immediate)**

These instructions are under [SVE Memory - 64-bit Gather](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	0	0	1	0	msz		0	1	imm5				1	U	ff	Pg			Zn			Zt								

Decode fields			Instruction Details
msz	U	ff	
00	0	0	<a href="#">LD1SB (vector plus immediate)</a>
00	0	1	<a href="#">LDFF1SB (vector plus immediate)</a>
00	1	0	<a href="#">LD1B (vector plus immediate)</a>
00	1	1	<a href="#">LDFF1B (vector plus immediate)</a>
01	0	0	<a href="#">LD1SH (vector plus immediate)</a>
01	0	1	<a href="#">LDFF1SH (vector plus immediate)</a>
01	1	0	<a href="#">LD1H (vector plus immediate)</a>
01	1	1	<a href="#">LDFF1H (vector plus immediate)</a>
10	0	0	<a href="#">LD1SW (vector plus immediate)</a>
10	0	1	<a href="#">LDFF1SW (vector plus immediate)</a>
10	1	0	<a href="#">LD1W (vector plus immediate)</a>
10	1	1	<a href="#">LDFF1W (vector plus immediate)</a>
11	0		UNALLOCATED
11	1	0	<a href="#">LD1D (vector plus immediate)</a>
11	1	1	<a href="#">LDFF1D (vector plus immediate)</a>

**SVE 64-bit gather load (scalar plus 64-bit unscaled offsets)**

These instructions are under [SVE Memory - 64-bit Gather](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	msz		1	0	Zm				1	U	ff	Pg			Rn				Zt						

Decode fields			Instruction Details
msz	U	ff	
00	0	0	<a href="#">LD1SB (scalar plus vector)</a>
00	0	1	<a href="#">LDFF1SB (scalar plus vector)</a>
00	1	0	<a href="#">LD1B (scalar plus vector)</a>
00	1	1	<a href="#">LDFF1B (scalar plus vector)</a>
01	0	0	<a href="#">LD1SH (scalar plus vector)</a>
01	0	1	<a href="#">LDFF1SH (scalar plus vector)</a>
01	1	0	<a href="#">LD1H (scalar plus vector)</a>
01	1	1	<a href="#">LDFF1H (scalar plus vector)</a>
10	0	0	<a href="#">LD1SW (scalar plus vector)</a>
10	0	1	<a href="#">LDFF1SW (scalar plus vector)</a>
10	1	0	<a href="#">LD1W (scalar plus vector)</a>
10	1	1	<a href="#">LDFF1W (scalar plus vector)</a>
11	0		UNALLOCATED
11	1	0	<a href="#">LD1D (scalar plus vector)</a>
11	1	1	<a href="#">LDFF1D (scalar plus vector)</a>

**SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets)**

These instructions are under [SVE Memory - 64-bit Gather](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	msz	xs	0	Zm				0	U	ff	Pg			Rn				Zt							

Decode fields			Instruction Details
msz	U	ff	
00	0	0	<a href="#">LD1SB (scalar plus vector)</a>
00	0	1	<a href="#">LDFF1SB (scalar plus vector)</a>
00	1	0	<a href="#">LD1B (scalar plus vector)</a>
00	1	1	<a href="#">LDFF1B (scalar plus vector)</a>
01	0	0	<a href="#">LD1SH (scalar plus vector)</a>
01	0	1	<a href="#">LDFF1SH (scalar plus vector)</a>
01	1	0	<a href="#">LD1H (scalar plus vector)</a>
01	1	1	<a href="#">LDFF1H (scalar plus vector)</a>
10	0	0	<a href="#">LD1SW (scalar plus vector)</a>
10	0	1	<a href="#">LDFF1SW (scalar plus vector)</a>
10	1	0	<a href="#">LD1W (scalar plus vector)</a>
10	1	1	<a href="#">LDFF1W (scalar plus vector)</a>
11	0		UNALLOCATED
11	1	0	<a href="#">LD1D (scalar plus vector)</a>
11	1	1	<a href="#">LDFF1D (scalar plus vector)</a>

## SVE Memory - Contiguous Store and Unsized Contiguous

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110010							op0			op1						0	op2		0						op3						

Decode fields				Instruction details
op0	op1	op2	op3	
0xx	00	0		<a href="#">SVE store multiple structures (quadwords, scalar plus immediate)</a>
0xx	01	0		UNALLOCATED
0xx	1x	0		<a href="#">SVE store multiple structures (quadwords, scalar plus scalar)</a>
10x		0		UNALLOCATED
110		0	0	<a href="#">STR (predicate)</a>
110		0	1	UNALLOCATED
110		1		<a href="#">STR (vector)</a>
111		0		UNALLOCATED
!= 110		1		<a href="#">SVE contiguous store (scalar plus scalar)</a>

## SVE store multiple structures (quadwords, scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Store and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	num	0	0	imm4			0 0 0			Pg			Rn			Zt								

Decode fields	Instruction Details	Feature
num		
00	UNALLOCATED	-
01	<a href="#">ST2Q (scalar plus immediate)</a>	FEAT_SVE2p1
10	<a href="#">ST3Q (scalar plus immediate)</a>	FEAT_SVE2p1
11	<a href="#">ST4Q (scalar plus immediate)</a>	FEAT_SVE2p1

## SVE store multiple structures (quadwords, scalar plus scalar)

These instructions are under [SVE Memory - Contiguous Store and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	num	1	Rm			0	0	0	Pg		Rn			Zt										

Decode fields num	Instruction Details	Feature
00	UNALLOCATED	-
01	<a href="#">ST2Q (scalar plus scalar)</a>	FEAT_SVE2p1
10	<a href="#">ST3Q (scalar plus scalar)</a>	FEAT_SVE2p1
11	<a href="#">ST4Q (scalar plus scalar)</a>	FEAT_SVE2p1

### SVE contiguous store (scalar plus scalar)

These instructions are under [SVE Memory - Contiguous Store and Unsized Contiguous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	!= 110		o2	Rm				0		1	0	Pg		Rn				Zt							
opc																															

The following constraints also apply to this encoding: `opc != 110 && opc != 110`

Decode fields		Instruction Details	Feature
opc	o2		
00x		<a href="#">ST1B (scalar plus scalar, single register)</a>	-
01x		<a href="#">ST1H (scalar plus scalar, single register)</a>	-
100	0	<a href="#">ST1W (scalar plus scalar, single register)</a> — <a href="#">SVE2</a>	FEAT_SVE2p1
101		<a href="#">ST1W (scalar plus scalar, single register)</a> — <a href="#">SVE</a>	-
111	0	<a href="#">ST1D (scalar plus scalar, single register)</a> — <a href="#">SVE2</a>	FEAT_SVE2p1
111	1	<a href="#">ST1D (scalar plus scalar, single register)</a> — <a href="#">SVE</a>	-

## SVE Memory - Non-temporal and Quadword Scatter Store

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110010								op0			op1						001														

Decode fields		Instruction details	Feature
op0	op1		
000	1	<a href="#">ST1Q</a>	FEAT_SVE2p1
!= 000	1	UNALLOCATED	-
xx0	0	<a href="#">SVE2 64-bit scatter non-temporal store (vector plus scalar)</a>	-
xx1	0	<a href="#">SVE2 32-bit scatter non-temporal store (vector plus scalar)</a>	-

### SVE2 64-bit scatter non-temporal store (vector plus scalar)

These instructions are under [SVE Memory - Non-temporal and Quadword Scatter Store](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz	0	0				Rm			0	0	1		Pg				Zn						Zt	

Decode fields msz	Instruction Details
00	<a href="#">STNT1B (vector plus scalar)</a>
01	<a href="#">STNT1H (vector plus scalar)</a>
10	<a href="#">STNT1W (vector plus scalar)</a>
11	<a href="#">STNT1D (vector plus scalar)</a>

**SVE2 32-bit scatter non-temporal store (vector plus scalar)**

These instructions are under [SVE Memory - Non-temporal and Quadword Scatter Store](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz	1	0				Rm			0	0	1		Pg				Zn					Zt		

Decode fields msz	Instruction Details
00	<a href="#">STNT1B (vector plus scalar)</a>
01	<a href="#">STNT1H (vector plus scalar)</a>
10	<a href="#">STNT1W (vector plus scalar)</a>
11	UNALLOCATED

**SVE Memory - Non-temporal and Multi-register Contiguous Store**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										op0										011											

Decode fields op0	Instruction details
00	<a href="#">SVE contiguous non-temporal store (scalar plus scalar)</a>
!= 00	<a href="#">SVE store multiple structures (scalar plus scalar)</a>

**SVE contiguous non-temporal store (scalar plus scalar)**

These instructions are under [SVE Memory - Non-temporal and Multi-register Contiguous Store](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz	0	0				Rm			0	1	1		Pg				Rn					Zt		

Decode fields msz	Instruction Details
00	<a href="#">STNT1B (scalar plus scalar, single register)</a>
01	<a href="#">STNT1H (scalar plus scalar, single register)</a>
10	<a href="#">STNT1W (scalar plus scalar, single register)</a>
11	<a href="#">STNT1D (scalar plus scalar, single register)</a>

**SVE store multiple structures (scalar plus scalar)**

These instructions are under [SVE Memory - Non-temporal and Multi-register Contiguous Store](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz	!= 00					Rm			0	1	1		Pg				Rn					Zt		
								opc																							

The following constraints also apply to this encoding: `opc != 00 && opc != 00`

Decode fields		Instruction Details
msz	opc	
00	01	<a href="#">ST2B (scalar plus scalar)</a>
00	10	<a href="#">ST3B (scalar plus scalar)</a>
00	11	<a href="#">ST4B (scalar plus scalar)</a>
01	01	<a href="#">ST2H (scalar plus scalar)</a>
01	10	<a href="#">ST3H (scalar plus scalar)</a>
01	11	<a href="#">ST4H (scalar plus scalar)</a>
10	01	<a href="#">ST2W (scalar plus scalar)</a>
10	10	<a href="#">ST3W (scalar plus scalar)</a>
10	11	<a href="#">ST4W (scalar plus scalar)</a>
11	01	<a href="#">ST2D (scalar plus scalar)</a>
11	10	<a href="#">ST3D (scalar plus scalar)</a>
11	11	<a href="#">ST4D (scalar plus scalar)</a>

## SVE Memory - Scatter with Optional Sign Extend

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110010								op0							1			0													

Decode fields		Instruction details
msz	opc	
00		<a href="#">SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets)</a>
01		<a href="#">SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets)</a>
10		<a href="#">SVE 32-bit scatter store (scalar plus 32-bit unscaled offsets)</a>
11		<a href="#">SVE 32-bit scatter store (scalar plus 32-bit scaled offsets)</a>

### SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets)

These instructions are under [SVE Memory - Scatter with Optional Sign Extend](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz		0	0	Zm				1	xs	0	Pg			Rn				Zt						

Decode fields		Instruction Details
msz	opc	
00		<a href="#">ST1B (scalar plus vector)</a>
01		<a href="#">ST1H (scalar plus vector)</a>
10		<a href="#">ST1W (scalar plus vector)</a>
11		<a href="#">ST1D (scalar plus vector)</a>

### SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets)

These instructions are under [SVE Memory - Scatter with Optional Sign Extend](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz		0	1	Zm				1	xs	0	Pg			Rn				Zt						

Decode fields		Instruction Details
msz	opc	
00		UNALLOCATED
01		<a href="#">ST1H (scalar plus vector)</a>

Decode fields msz	Instruction Details
10	<a href="#">ST1W (scalar plus vector)</a>
11	<a href="#">ST1D (scalar plus vector)</a>

**SVE 32-bit scatter store (scalar plus 32-bit unscaled offsets)**

These instructions are under [SVE Memory - Scatter with Optional Sign Extend](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz	1	0		Zm		1	xs	0		Pg			Rn							Zt				

Decode fields msz	Instruction Details
00	<a href="#">ST1B (scalar plus vector)</a>
01	<a href="#">ST1H (scalar plus vector)</a>
10	<a href="#">ST1W (scalar plus vector)</a>
11	UNALLOCATED

**SVE 32-bit scatter store (scalar plus 32-bit scaled offsets)**

These instructions are under [SVE Memory - Scatter with Optional Sign Extend](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz	1	1		Zm		1	xs	0		Pg			Rn							Zt				

Decode fields msz	Instruction Details
00	UNALLOCATED
01	<a href="#">ST1H (scalar plus vector)</a>
10	<a href="#">ST1W (scalar plus vector)</a>
11	UNALLOCATED

**SVE Memory - Scatter**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										op0																					

Decode fields op0	Instruction details
00	<a href="#">SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets)</a>
01	<a href="#">SVE 64-bit scatter store (scalar plus 64-bit scaled offsets)</a>
10	<a href="#">SVE 64-bit scatter store (vector plus immediate)</a>
11	<a href="#">SVE 32-bit scatter store (vector plus immediate)</a>

**SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets)**

These instructions are under [SVE Memory - Scatter](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz	0	0		Zm		1	0	1		Pg			Rn							Zt				



Decode fields msz	Instruction Details
00	<a href="#">ST1B (scalar plus vector)</a>
01	<a href="#">ST1H (scalar plus vector)</a>
10	<a href="#">ST1W (scalar plus vector)</a>
11	<a href="#">ST1D (scalar plus vector)</a>

**SVE 64-bit scatter store (scalar plus 64-bit scaled offsets)**

These instructions are under [SVE Memory - Scatter](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz	0	1			Zm				1	0	1		Pg				Rn					Zt		

Decode fields msz	Instruction Details
00	UNALLOCATED
01	<a href="#">ST1H (scalar plus vector)</a>
10	<a href="#">ST1W (scalar plus vector)</a>
11	<a href="#">ST1D (scalar plus vector)</a>

**SVE 64-bit scatter store (vector plus immediate)**

These instructions are under [SVE Memory - Scatter](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz	1	0			imm5				1	0	1		Pg				Zn					Zt		

Decode fields msz	Instruction Details
00	<a href="#">ST1B (vector plus immediate)</a>
01	<a href="#">ST1H (vector plus immediate)</a>
10	<a href="#">ST1W (vector plus immediate)</a>
11	<a href="#">ST1D (vector plus immediate)</a>

**SVE 32-bit scatter store (vector plus immediate)**

These instructions are under [SVE Memory - Scatter](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz	1	1			imm5				1	0	1		Pg				Zn					Zt		

Decode fields msz	Instruction Details
00	<a href="#">ST1B (vector plus immediate)</a>
01	<a href="#">ST1H (vector plus immediate)</a>
10	<a href="#">ST1W (vector plus immediate)</a>
11	UNALLOCATED

**SVE Memory - Contiguous Store with Immediate Offset**

These instructions are under [SVE encodings](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										op0	op1																				

Decode fields		Instruction details
op0	op1	
00	1	<a href="#">SVE contiguous non-temporal store (scalar plus immediate)</a>
!= 00	1	<a href="#">SVE store multiple structures (scalar plus immediate)</a>
	0	<a href="#">SVE contiguous store (scalar plus immediate)</a>

**SVE contiguous non-temporal store (scalar plus immediate)**

These instructions are under [SVE Memory - Contiguous Store with Immediate Offset](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz	0	0	1	imm4			1	1	1	Pg			Rn			Zt								

Decode fields		Instruction Details
msz		
00		<a href="#">STNT1B (scalar plus immediate, single register)</a>
01		<a href="#">STNT1H (scalar plus immediate, single register)</a>
10		<a href="#">STNT1W (scalar plus immediate, single register)</a>
11		<a href="#">STNT1D (scalar plus immediate, single register)</a>

**SVE store multiple structures (scalar plus immediate)**

These instructions are under [SVE Memory - Contiguous Store with Immediate Offset](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz	!= 00	1	imm4			1	1	1	Pg		Rn				Zt									
opc																															

The following constraints also apply to this encoding: opc != 00 && opc != 00

Decode fields		Instruction Details
msz	opc	
00	01	<a href="#">ST2B (scalar plus immediate)</a>
00	10	<a href="#">ST3B (scalar plus immediate)</a>
00	11	<a href="#">ST4B (scalar plus immediate)</a>
01	01	<a href="#">ST2H (scalar plus immediate)</a>
01	10	<a href="#">ST3H (scalar plus immediate)</a>
01	11	<a href="#">ST4H (scalar plus immediate)</a>
10	01	<a href="#">ST2W (scalar plus immediate)</a>
10	10	<a href="#">ST3W (scalar plus immediate)</a>
10	11	<a href="#">ST4W (scalar plus immediate)</a>
11	01	<a href="#">ST2D (scalar plus immediate)</a>
11	10	<a href="#">ST3D (scalar plus immediate)</a>
11	11	<a href="#">ST4D (scalar plus immediate)</a>

**SVE contiguous store (scalar plus immediate)**

These instructions are under [SVE Memory - Contiguous Store with Immediate Offset](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz	opc	0		imm4	1	1	1		Pg									Rn					Zt	

Decode fields		Instruction Details	Feature
msz	opc		
00		<a href="#">ST1B (scalar plus immediate, single register)</a>	-
01		<a href="#">ST1H (scalar plus immediate, single register)</a>	-
10	00	<a href="#">ST1W (scalar plus immediate, single register)</a> — <a href="#">SVE2</a>	FEAT_SVE2p1
10	01	UNALLOCATED	-
10	1x	<a href="#">ST1W (scalar plus immediate, single register)</a> — <a href="#">SVE</a>	-
11	0x	UNALLOCATED	-
11	10	<a href="#">ST1D (scalar plus immediate, single register)</a> — <a href="#">SVE2</a>	FEAT_SVE2p1
11	11	<a href="#">ST1D (scalar plus immediate, single register)</a> — <a href="#">SVE</a>	-

## Data Processing -- Immediate

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					100				op0																						

Decode fields		Instruction details
op0		
00xx		<a href="#">PC-rel. addressing</a>
010x		<a href="#">Add/subtract (immediate)</a>
0110		<a href="#">Add/subtract (immediate, with tags)</a>
0111		<a href="#">Min/max (immediate)</a>
100x		<a href="#">Logical (immediate)</a>
101x		<a href="#">Move wide (immediate)</a>
110x		<a href="#">Bitfield</a>
111x		<a href="#">Extract</a>

## PC-rel. addressing

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op	immlo	1	0	0	0	0																									Rd
																immhi															

Decode fields		Instruction Details
op		
0		<a href="#">ADR</a>
1		<a href="#">ADRP</a>

## Add/subtract (immediate)

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	0	0	0	1	0	sh	imm12												Rn				Rd					

Decode fields			Instruction Details
sf	op	S	
0	0	0	<a href="#">ADD (immediate)</a> — 32-bit
0	0	1	<a href="#">ADDS (immediate)</a> — 32-bit
0	1	0	<a href="#">SUB (immediate)</a> — 32-bit
0	1	1	<a href="#">SUBS (immediate)</a> — 32-bit

Decode fields			Instruction Details
sf	op	S	
1	0	0	<a href="#">ADD (immediate) — 64-bit</a>
1	0	1	<a href="#">ADDS (immediate) — 64-bit</a>
1	1	0	<a href="#">SUB (immediate) — 64-bit</a>
1	1	1	<a href="#">SUBS (immediate) — 64-bit</a>

**Add/subtract (immediate, with tags)**

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	0	0	0	1	1	0	uimm6						op3	uimm4				Rn				Rd						

Decode fields			Instruction Details	Feature
sf	op	S		
0			UNALLOCATED	-
1		1	UNALLOCATED	-
1	0	0	<a href="#">ADDG</a>	FEAT_MTE
1	1	0	<a href="#">SUBG</a>	FEAT_MTE

**Min/max (immediate)**

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	0	0	0	1	1	1	opc				imm8						Rn				Rd							

Decode fields				Instruction Details	Feature
sf	op	S	opc		
	0		01xx	UNALLOCATED	-
	0		1xxx	UNALLOCATED	-
	0	1	00xx	UNALLOCATED	-
	1			UNALLOCATED	-
0	0	0	0000	<a href="#">SMAX (immediate) — 32-bit</a>	FEAT_CSSC
0	0	0	0001	<a href="#">UMAX (immediate) — 32-bit</a>	FEAT_CSSC
0	0	0	0010	<a href="#">SMIN (immediate) — 32-bit</a>	FEAT_CSSC
0	0	0	0011	<a href="#">UMIN (immediate) — 32-bit</a>	FEAT_CSSC
1	0	0	0000	<a href="#">SMAX (immediate) — 64-bit</a>	FEAT_CSSC
1	0	0	0001	<a href="#">UMAX (immediate) — 64-bit</a>	FEAT_CSSC
1	0	0	0010	<a href="#">SMIN (immediate) — 64-bit</a>	FEAT_CSSC
1	0	0	0011	<a href="#">UMIN (immediate) — 64-bit</a>	FEAT_CSSC

**Logical (immediate)**

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	opc		1	0	0	1	0	0	N	immr						imms				Rn				Rd							

Decode fields			Instruction Details
sf	opc	N	
0		1	UNALLOCATED
0	00	0	<a href="#">AND (immediate) — 32-bit</a>

Decode fields			Instruction Details
sf	opc	N	
0	01	0	<a href="#">ORR (immediate) — 32-bit</a>
0	10	0	<a href="#">EOR (immediate) — 32-bit</a>
0	11	0	<a href="#">ANDS (immediate) — 32-bit</a>
1	00		<a href="#">AND (immediate) — 64-bit</a>
1	01		<a href="#">ORR (immediate) — 64-bit</a>
1	10		<a href="#">EOR (immediate) — 64-bit</a>
1	11		<a href="#">ANDS (immediate) — 64-bit</a>

### Move wide (immediate)

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	opc	1	0	0	1	0	1	hw	imm16																		Rd				

Decode fields			Instruction Details
sf	opc	hw	
	01		UNALLOCATED
0		1x	UNALLOCATED
0	00	0x	<a href="#">MOVN — 32-bit</a>
0	10	0x	<a href="#">MOVZ — 32-bit</a>
0	11	0x	<a href="#">MOVK — 32-bit</a>
1	00		<a href="#">MOVN — 64-bit</a>
1	10		<a href="#">MOVZ — 64-bit</a>
1	11		<a href="#">MOVK — 64-bit</a>

### Bitfield

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	opc	1	0	0	1	1	0	N	immr				imms				Rn				Rd										

Decode fields			Instruction Details
sf	opc	N	
	11		UNALLOCATED
0		1	UNALLOCATED
0	00	0	<a href="#">SBFM — 32-bit</a>
0	01	0	<a href="#">BFM — 32-bit</a>
0	10	0	<a href="#">UBFM — 32-bit</a>
1		0	UNALLOCATED
1	00	1	<a href="#">SBFM — 64-bit</a>
1	01	1	<a href="#">BFM — 64-bit</a>
1	10	1	<a href="#">UBFM — 64-bit</a>

### Extract

These instructions are under [Data Processing -- Immediate](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op21	1	0	0	1	1	1	N	o0	Rm				imms				Rn				Rd									

Decode fields					Instruction Details
sf	op21	N	o0	imms	
	x1				UNALLOCATED
	00		1		UNALLOCATED
	1x				UNALLOCATED
0				1xxxxx	UNALLOCATED
0		1			UNALLOCATED
0	00	0	0	0xxxxx	<a href="#">EXTR — 32-bit</a>
1		0			UNALLOCATED
1	00	1	0		<a href="#">EXTR — 64-bit</a>

## Branches, Exception Generating and System instructions

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0				101		op1																				op2					

Decode fields			Instruction details
op0	op1	op2	
010	0xxxxxxxxxxxxx		<a href="#">Conditional branch (immediate)</a>
110	00xxxxxxxxxxxxx		<a href="#">Exception generation</a>
110	01000000110001		<a href="#">System instructions with register argument</a>
110	01000000110010	11111	<a href="#">Hints</a>
110	01000000110011		<a href="#">Barriers</a>
110	0100000xxx0100		<a href="#">PSTATE</a>
110	0100100xxxxxxx		<a href="#">System with result</a>
110	0100x01xxxxxxx		<a href="#">System instructions</a>
110	0100x1xxxxxxx		<a href="#">System register move</a>
110	0101x01xxxxxxx		<a href="#">System pair instructions</a>
110	0101x1xxxxxxx		<a href="#">System register pair move</a>
110	1xxxxxxxxxxxxx		<a href="#">Unconditional branch (register)</a>
x00			<a href="#">Unconditional branch (immediate)</a>
x01	0xxxxxxxxxxxxx		<a href="#">Compare and branch (immediate)</a>
x01	1xxxxxxxxxxxxx		<a href="#">Test and branch (immediate)</a>

### Conditional branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	o1	imm19																	o0		cond				

Decode fields		Instruction Details	Feature
o1	o0		
0	0	<a href="#">B.cond</a>	-
0	1	<a href="#">BC.cond</a>	FEAT_HBC
1		UNALLOCATED	-

### Exception generation

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	opc				imm16																op2		LL	

Decode fields			Instruction Details	Feature
opc	op2	LL		
	001		UNALLOCATED	-
	01x		UNALLOCATED	-
	1xx		UNALLOCATED	-
000	000	00	UNALLOCATED	-
000	000	01	<a href="#">SVC</a>	-
000	000	10	<a href="#">HVC</a>	-
000	000	11	<a href="#">SMC</a>	-
001	000	x1	UNALLOCATED	-
001	000	00	<a href="#">BRK</a>	-
001	000	1x	UNALLOCATED	-
010	000	x1	UNALLOCATED	-
010	000	00	<a href="#">HLT</a>	-
010	000	1x	UNALLOCATED	-
011	000	00	<a href="#">TCANCEL</a>	FEAT_TME
011	000	01	UNALLOCATED	-
011	000	1x	UNALLOCATED	-
100	000		UNALLOCATED	-
101	000	00	UNALLOCATED	-
101	000	01	<a href="#">DCPS1</a>	-
101	000	10	<a href="#">DCPS2</a>	-
101	000	11	<a href="#">DCPS3</a>	-
110	000		UNALLOCATED	-
111	000		UNALLOCATED	-

### System instructions with register argument

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	0	1	CRm				op2		Rt					

Decode fields		Instruction Details	Feature
CRm	op2		
!= 0000		UNALLOCATED	-
0000	000	<a href="#">WFET</a>	FEAT_WFxT
0000	001	<a href="#">WFIT</a>	FEAT_WFxT
0000	01x	UNALLOCATED	-
0000	1xx	UNALLOCATED	-

### Hints

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	CRm				op2		1	1	1	1	1	

Decode fields		Instruction Details	Feature
CRm	op2		
		<a href="#">HINT</a>	-

Decode fields		Instruction Details	Feature
CRm	op2		
0000	000	<a href="#">NOP</a>	-
0000	001	<a href="#">YIELD</a>	-
0000	010	<a href="#">WFE</a>	-
0000	011	<a href="#">WFI</a>	-
0000	100	<a href="#">SEV</a>	-
0000	101	<a href="#">SEVL</a>	-
0000	110	<a href="#">DGH</a>	FEAT_DGH
0000	111	<a href="#">XPACD, XPACL, XPACLRI</a>	FEAT_PAuth
0001	000	<a href="#">PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA</a> — <a href="#">PACIA1716</a>	FEAT_PAuth
0001	010	<a href="#">PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB</a> — <a href="#">PACIB1716</a>	FEAT_PAuth
0001	100	<a href="#">AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA</a> — <a href="#">AUTIA1716</a>	FEAT_PAuth
0001	110	<a href="#">AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB</a> — <a href="#">AUTIB1716</a>	FEAT_PAuth
0010	000	<a href="#">ESB</a>	FEAT_RAS
0010	001	<a href="#">PSB CSYNC</a>	FEAT_SPE
0010	010	<a href="#">TSB CSYNC</a>	FEAT_TRF
0010	011	<a href="#">GCSB DSYNC</a>	FEAT_GCS
0010	100	<a href="#">CSDB</a>	-
0010	110	<a href="#">CLRBHB</a>	FEAT_CLRBHB
0011	000	<a href="#">PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA</a> — <a href="#">PACIAZ</a>	FEAT_PAuth
0011	001	<a href="#">PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA</a> — <a href="#">PACIASP</a>	FEAT_PAuth
0011	010	<a href="#">PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB</a> — <a href="#">PACIBZ</a>	FEAT_PAuth
0011	011	<a href="#">PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB</a> — <a href="#">PACIBSP</a>	FEAT_PAuth
0011	100	<a href="#">AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA</a> — <a href="#">AUTIAZ</a>	FEAT_PAuth
0011	101	<a href="#">AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA</a> — <a href="#">AUTIASP</a>	FEAT_PAuth
0011	110	<a href="#">AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB</a> — <a href="#">AUTIBZ</a>	FEAT_PAuth
0011	111	<a href="#">AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB</a> — <a href="#">AUTIBSP</a>	FEAT_PAuth
0100	xx0	<a href="#">BTI</a>	FEAT_BTI
0101	000	<a href="#">CHKFEAT</a>	FEAT_CHK

## Barriers

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	CRm			op2			Rt					

Decode fields		Rt	Instruction Details	Feature
CRm	op2			
	000		UNALLOCATED	-
	001	<b>!= 11111</b>	UNALLOCATED	-
	010	11111	<a href="#">CLREX</a>	-
	100	11111	<a href="#">DSB</a> — <a href="#">memory barrier</a>	-
	101	11111	<a href="#">DMB</a>	-
	110	11111	<a href="#">ISB</a>	-
	111	<b>!= 11111</b>	UNALLOCATED	-
	111	11111	<a href="#">SB</a>	FEAT_SB
xx0x	001	11111	UNALLOCATED	-
xx10	001	11111	<a href="#">DSB</a> — <a href="#">Memory nXS barrier</a>	FEAT_XS
xx11	001	11111	UNALLOCATED	-



Decode fields			Instruction Details	Feature
CRm	op2	Rt		
0000	011	11111	<a href="#">TCOMMIT</a>	FEAT_TME
0001	011		UNALLOCATED	-
001x	011		UNALLOCATED	-
01xx	011		UNALLOCATED	-
1xxx	011		UNALLOCATED	-

## PSTATE

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	op1			0	1	0	0	CRm			op2			Rt					

Decode fields			Instruction Details	Feature
op1	op2	Rt		
		!= 11111	UNALLOCATED	-
		11111	<a href="#">MSR (immediate)</a>	-
000	000	11111	<a href="#">CFINV</a>	FEAT_FlagM
000	001	11111	<a href="#">XAFLAG</a>	FEAT_FlagM2
000	010	11111	<a href="#">AXFLAG</a>	FEAT_FlagM2

## System with result

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	1	0	0	op1			CRn			CRm			op2			Rt						

Decode fields				Instruction Details	Feature
op1	CRn	CRm	op2		
!= 011				UNALLOCATED	-
011	!= 0011			UNALLOCATED	-
011	0011		!= 011	UNALLOCATED	-
011	0011	!= 000x	011	UNALLOCATED	-
011	0011	0000	011	<a href="#">TSTART</a>	FEAT_TME
011	0011	0001	011	<a href="#">TTEST</a>	FEAT_TME

## System instructions

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	L	0	1	op1			CRn			CRm			op2			Rt						

Decode fields	Instruction Details
L	
0	<a href="#">SYS</a>
1	<a href="#">SYSL</a>

## System register move

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	L	1	o0	op1			CRn			CRm			op2			Rt						

### Decode fields

L	Instruction Details
0	<a href="#">MSR (register)</a>
1	<a href="#">MRS</a>

## System pair instructions

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	1	L	0	1	op1			CRn			CRm			op2			Rt						

### Decode fields

L	Instruction Details	Feature
0	<a href="#">SYSP</a>	FEAT_SYSINSTR128
1	UNALLOCATED	-

## System register pair move

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	1	L	1	o0	op1			CRn			CRm			op2			Rt						

### Decode fields

L	Instruction Details	Feature
0	<a href="#">MSRR</a>	FEAT_SYSREG128
1	<a href="#">MRRS</a>	FEAT_SYSREG128

## Unconditional branch (register)

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	opc			op2			op3			Rn			op4												

Decode fields				Instruction Details			Feature
opc	op2	op3	Rn	op4			
	!= 11111				UNALLOCATED		-
0000	11111	000000		!= 00000	UNALLOCATED		-
0000	11111	000000		00000	<a href="#">BR</a>		-
0000	11111	000001			UNALLOCATED		-
0000	11111	000010		!= 11111	UNALLOCATED		-
0000	11111	000010		11111	<a href="#">BRAA, BRAAZ, BRAB, BRABZ</a> — <a href="#">key A, zero modifier</a>		FEAT_PAuth
0000	11111	000011		!= 11111	UNALLOCATED		-
0000	11111	000011		11111	<a href="#">BRAA, BRAAZ, BRAB, BRABZ</a> — <a href="#">key B, zero modifier</a>		FEAT_PAuth
0000	11111	0001xx			UNALLOCATED		-

opc	Decode fields			Rn	op4	Instruction Details	Feature
	op2	op3					
0000	11111	001xxx				UNALLOCATED	-
0000	11111	01xxxx				UNALLOCATED	-
0000	11111	1xxxxx				UNALLOCATED	-
0001	11111	000000		!= 00000		UNALLOCATED	-
0001	11111	000000		00000	<a href="#">BLR</a>		-
0001	11111	000001				UNALLOCATED	-
0001	11111	000010		!= 11111		UNALLOCATED	-
0001	11111	000010		11111	<a href="#">BLRAA, BLRAAZ, BLRAB, BLRABZ</a> — <a href="#">key A, zero modifier</a>		FEAT_PAuth
0001	11111	000011		!= 11111		UNALLOCATED	-
0001	11111	000011		11111	<a href="#">BLRAA, BLRAAZ, BLRAB, BLRABZ</a> — <a href="#">key B, zero modifier</a>		FEAT_PAuth
0001	11111	0001xx				UNALLOCATED	-
0001	11111	001xxx				UNALLOCATED	-
0001	11111	01xxxx				UNALLOCATED	-
0001	11111	1xxxxx				UNALLOCATED	-
0010	11111	000000		!= 00000		UNALLOCATED	-
0010	11111	000000		00000	<a href="#">RET</a>		-
0010	11111	000001				UNALLOCATED	-
0010	11111	000010	!= 11111	!= 11111		UNALLOCATED	-
0010	11111	000010	!= 11111	11111		UNALLOCATED	-
0010	11111	000010	11111	!= 11111		UNALLOCATED	-
0010	11111	000010	11111	11111	<a href="#">RETAA, RETAB</a> — <a href="#">RETAA</a>		FEAT_PAuth
0010	11111	000011	!= 11111	!= 11111		UNALLOCATED	-
0010	11111	000011	!= 11111	11111		UNALLOCATED	-
0010	11111	000011	11111	!= 11111		UNALLOCATED	-
0010	11111	000011	11111	11111	<a href="#">RETAA, RETAB</a> — <a href="#">RETAB</a>		FEAT_PAuth
0010	11111	0001xx				UNALLOCATED	-
0010	11111	001xxx				UNALLOCATED	-
0010	11111	01xxxx				UNALLOCATED	-
0010	11111	1xxxxx				UNALLOCATED	-
0011	11111					UNALLOCATED	-
0100	11111	000000	!= 11111	!= 00000		UNALLOCATED	-
0100	11111	000000	!= 11111	00000		UNALLOCATED	-
0100	11111	000000	11111	!= 00000		UNALLOCATED	-
0100	11111	000000	11111	00000	<a href="#">ERET</a>		-

Decode fields					Instruction Details	Feature
opc	op2	op3	Rn	op4		
0100	11111	000001			UNALLOCATED	-
0100	11111	000010	!= 11111	!= 11111	UNALLOCATED	-
0100	11111	000010	!= 11111	11111	UNALLOCATED	-
0100	11111	000010	11111	!= 11111	UNALLOCATED	-
0100	11111	000010	11111	11111	<a href="#">ERETAA, ERETAB — ERETAA</a>	FEAT_PAuth
0100	11111	000011	!= 11111	!= 11111	UNALLOCATED	-
0100	11111	000011	!= 11111	11111	UNALLOCATED	-
0100	11111	000011	11111	!= 11111	UNALLOCATED	-
0100	11111	000011	11111	11111	<a href="#">ERETAA, ERETAB — ERETAB</a>	FEAT_PAuth
0100	11111	0001xx			UNALLOCATED	-
0100	11111	001xxx			UNALLOCATED	-
0100	11111	01xxxx			UNALLOCATED	-
0100	11111	1xxxxxx			UNALLOCATED	-
0101	11111	!= 000000			UNALLOCATED	-
0101	11111	000000	!= 11111	!= 00000	UNALLOCATED	-
0101	11111	000000	!= 11111	00000	UNALLOCATED	-
0101	11111	000000	11111	!= 00000	UNALLOCATED	-
0101	11111	000000	11111	00000	<a href="#">DRPS</a>	-
011x	11111				UNALLOCATED	-
1000	11111	00000x			UNALLOCATED	-
1000	11111	000010			<a href="#">BRAA, BRAAZ, BRAB, BRABZ — key A, register modifier</a>	FEAT_PAuth
1000	11111	000011			<a href="#">BRAA, BRAAZ, BRAB, BRABZ — key B, register modifier</a>	FEAT_PAuth
1000	11111	0001xx			UNALLOCATED	-
1000	11111	001xxx			UNALLOCATED	-
1000	11111	01xxxx			UNALLOCATED	-
1000	11111	1xxxxxx			UNALLOCATED	-
1001	11111	00000x			UNALLOCATED	-
1001	11111	000010			<a href="#">BLRAA, BLRAAZ, BLRAB, BLRABZ — key A, register modifier</a>	FEAT_PAuth
1001	11111	000011			<a href="#">BLRAA, BLRAAZ, BLRAB, BLRABZ — key B, register modifier</a>	FEAT_PAuth
1001	11111	0001xx			UNALLOCATED	-
1001	11111	001xxx			UNALLOCATED	-
1001	11111	01xxxx			UNALLOCATED	-
1001	11111	1xxxxxx			UNALLOCATED	-
101x	11111				UNALLOCATED	-
11xx	11111				UNALLOCATED	-

**Unconditional branch (immediate)**

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
op	0	0	1	0	1	imm26																										

Decode fields	Instruction Details
op	
0	<a href="#">B</a>
1	<a href="#">BL</a>

**Compare and branch (immediate)**

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	1	1	0	1	0	op	imm19																	Rt						

Decode fields		Instruction Details
sf	op	
0	0	<a href="#">CBZ — 32-bit</a>
0	1	<a href="#">CBNZ — 32-bit</a>
1	0	<a href="#">CBZ — 64-bit</a>
1	1	<a href="#">CBNZ — 64-bit</a>

**Test and branch (immediate)**

These instructions are under [Branches, Exception Generating and System instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
b5	0	1	1	0	1	1	op	b40					imm14															Rt			

Decode fields	Instruction Details
op	
0	<a href="#">TBZ</a>
1	<a href="#">TBNZ</a>

**Loads and Stores**

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0				1	op1	0	op2																								

Decode fields			Instruction details
op0	op1	op2	
0x00	0	00x1xxxxxxxxxxxx	<a href="#">Compare and swap pair</a>
0x00	1	00x000000xxxxxx	<a href="#">Advanced SIMD load/store multiple structures</a>
0x00	1	00xxxxxx1xxxxxx	UNALLOCATED
0x00	1	01x0xxxxxxxxxxxx	<a href="#">Advanced SIMD load/store multiple structures (post-indexed)</a>
0x00	1	0xx1xxxxxxxxxxxx	UNALLOCATED
0x00	1	10xx0000xxxxxx	<a href="#">Advanced SIMD load/store single structure</a>
0x00	1	11xxxxxxxxxxxxxx	<a href="#">Advanced SIMD load/store single structure (post-indexed)</a>
0x00	1	x0xx1xxxxxxxxxx	UNALLOCATED
0x00	1	x0xxx1xxxxxxxxxx	UNALLOCATED

0x00	1	x0xxxx1xxxxxxxx	UNALLOCATED
0x00	1	x0xxxx1xxxxxxxx	UNALLOCATED
0x01	0	1xx1xxxxx000010	<a href="#">RCW compare and swap</a>
0x01	0	1xx1xxxxx000011	<a href="#">RCW compare and swap pair</a>
0x01	0	1xx1xxxxxxxxxxx00	<a href="#">128-bit atomic memory operations</a>
1101	0	1000111110xxx11	<a href="#">GCS load/store</a>
1101	0	1xx1xxxxxxxxxxx	<a href="#">Load/store memory tags</a>
1x00	0	00x1xxxxxxxxxxx	<a href="#">Load/store exclusive pair</a>
1x00	1		UNALLOCATED
xx00	0	00x0xxxxxxxxxxx	<a href="#">Load/store exclusive register</a>
xx00	0	01x0xxxxxxxxxxx	<a href="#">Load/store ordered</a>
xx00	0	01x1xxxxxxxxxxx	<a href="#">Compare and swap</a>
xx01	0	10x0xxxxxxxxx10	<a href="#">LDIAPP/STILP</a>
xx01	0	11x000000000010	<a href="#">LDAPR/STLR (writeback)</a>
xx01	0	1xx0xxxxxxxxx00	<a href="#">LDAPR/STLR (unscaled immediate)</a>
xx01	1	1xx0xxxxxxxxx10	<a href="#">LDAPR/STLR (SIMD&amp;FP)</a>
xx01		0xxxxxxxxxxxxxx	<a href="#">Load register (literal)</a>
xx01		1xx0xxxxxxxxx01	<a href="#">Memory Copy and Memory Set</a>
xx10		00xxxxxxxxxxxxxx	<a href="#">Load/store no-allocate pair (offset)</a>
xx10		01xxxxxxxxxxxxxx	<a href="#">Load/store register pair (post-indexed)</a>
xx10		10xxxxxxxxxxxxxx	<a href="#">Load/store register pair (offset)</a>
xx10		11xxxxxxxxxxxxxx	<a href="#">Load/store register pair (pre-indexed)</a>
xx11		0xx0xxxxxxxxx00	<a href="#">Load/store register (unscaled immediate)</a>
xx11		0xx0xxxxxxxxx01	<a href="#">Load/store register (immediate post-indexed)</a>
xx11		0xx0xxxxxxxxx10	<a href="#">Load/store register (unprivileged)</a>
xx11		0xx0xxxxxxxxx11	<a href="#">Load/store register (immediate pre-indexed)</a>
xx11		0xx1xxxxxxxxx00	<a href="#">Atomic memory operations</a>
xx11		0xx1xxxxxxxxx10	<a href="#">Load/store register (register offset)</a>
xx11		0xx1xxxxxxxxx1	<a href="#">Load/store register (pac)</a>
xx11		1xxxxxxxxxxxxxx	<a href="#">Load/store register (unsigned immediate)</a>

## Compare and swap pair

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	sz	0	0	1	0	0	0	0	L	1	Rs			o0		Rt2			Rn			Rt									

Decode fields				Instruction Details		Feature
sz	L	o0	Rt2			
			!= 11111	UNALLOCATED		-
0	0	0	11111	<a href="#">CASE, CASPA, CASPAL, CASPL — 32-bit CASP</a>		FEAT_LSE
0	0	1	11111	<a href="#">CASP, CASPA, CASPAL, CASPL — 32-bit CASPL</a>		FEAT_LSE
0	1	0	11111	<a href="#">CASE, CASPA, CASPAL, CASPL — 32-bit CASPA</a>		FEAT_LSE
0	1	1	11111	<a href="#">CASP, CASPA, CASPAL, CASPL — 32-bit CASPAL</a>		FEAT_LSE
1	0	0	11111	<a href="#">CASE, CASPA, CASPAL, CASPL — 64-bit CASP</a>		FEAT_LSE
1	0	1	11111	<a href="#">CASP, CASPA, CASPAL, CASPL — 64-bit CASPL</a>		FEAT_LSE
1	1	0	11111	<a href="#">CASE, CASPA, CASPAL, CASPL — 64-bit CASPA</a>		FEAT_LSE
1	1	1	11111	<a href="#">CASP, CASPA, CASPAL, CASPL — 64-bit CASPAL</a>		FEAT_LSE

**Advanced SIMD load/store multiple structures**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	L	0	0	0	0	0	0	opcode				size			Rn				Rt				

Decode fields		Instruction Details
L	opcode	
0	0000	<a href="#">ST4 (multiple structures)</a>
0	0001	UNALLOCATED
0	0010	<a href="#">ST1 (multiple structures)</a> — <a href="#">four registers</a>
0	0011	UNALLOCATED
0	0100	<a href="#">ST3 (multiple structures)</a>
0	0101	UNALLOCATED
0	0110	<a href="#">ST1 (multiple structures)</a> — <a href="#">three registers</a>
0	0111	<a href="#">ST1 (multiple structures)</a> — <a href="#">one register</a>
0	1000	<a href="#">ST2 (multiple structures)</a>
0	1001	UNALLOCATED
0	1010	<a href="#">ST1 (multiple structures)</a> — <a href="#">two registers</a>
0	1011	UNALLOCATED
0	11xx	UNALLOCATED
1	0000	<a href="#">LD4 (multiple structures)</a>
1	0001	UNALLOCATED
1	0010	<a href="#">LD1 (multiple structures)</a> — <a href="#">four registers</a>
1	0011	UNALLOCATED
1	0100	<a href="#">LD3 (multiple structures)</a>
1	0101	UNALLOCATED
1	0110	<a href="#">LD1 (multiple structures)</a> — <a href="#">three registers</a>
1	0111	<a href="#">LD1 (multiple structures)</a> — <a href="#">one register</a>
1	1000	<a href="#">LD2 (multiple structures)</a>
1	1001	UNALLOCATED
1	1010	<a href="#">LD1 (multiple structures)</a> — <a href="#">two registers</a>
1	1011	UNALLOCATED
1	11xx	UNALLOCATED

**Advanced SIMD load/store multiple structures (post-indexed)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	L	0	Rm				opcode				size			Rn				Rt					

Decode fields		Instruction Details
L	Rm opcode	
0		UNALLOCATED
0		UNALLOCATED
0		UNALLOCATED
0		UNALLOCATED
0		UNALLOCATED
0		UNALLOCATED
0	!= 11111	<a href="#">ST4 (multiple structures)</a> — <a href="#">register offset</a>
0	!= 11111	<a href="#">ST1 (multiple structures)</a> — <a href="#">four registers, register offset</a>

Decode fields			Instruction Details
L	Rm	opcode	
0	!= 11111	0100	<a href="#">ST3 (multiple structures)</a> — <a href="#">register offset</a>
0	!= 11111	0110	<a href="#">ST1 (multiple structures)</a> — <a href="#">three registers, register offset</a>
0	!= 11111	0111	<a href="#">ST1 (multiple structures)</a> — <a href="#">one register, register offset</a>
0	!= 11111	1000	<a href="#">ST2 (multiple structures)</a> — <a href="#">register offset</a>
0	!= 11111	1010	<a href="#">ST1 (multiple structures)</a> — <a href="#">two registers, register offset</a>
0	11111	0000	<a href="#">ST4 (multiple structures)</a> — <a href="#">immediate offset</a>
0	11111	0010	<a href="#">ST1 (multiple structures)</a> — <a href="#">four registers, immediate offset</a>
0	11111	0100	<a href="#">ST3 (multiple structures)</a> — <a href="#">immediate offset</a>
0	11111	0110	<a href="#">ST1 (multiple structures)</a> — <a href="#">three registers, immediate offset</a>
0	11111	0111	<a href="#">ST1 (multiple structures)</a> — <a href="#">one register, immediate offset</a>
0	11111	1000	<a href="#">ST2 (multiple structures)</a> — <a href="#">immediate offset</a>
0	11111	1010	<a href="#">ST1 (multiple structures)</a> — <a href="#">two registers, immediate offset</a>
1		0001	UNALLOCATED
1		0011	UNALLOCATED
1		0101	UNALLOCATED
1		1001	UNALLOCATED
1		1011	UNALLOCATED
1		11xx	UNALLOCATED
1	!= 11111	0000	<a href="#">LD4 (multiple structures)</a> — <a href="#">register offset</a>
1	!= 11111	0010	<a href="#">LD1 (multiple structures)</a> — <a href="#">four registers, register offset</a>
1	!= 11111	0100	<a href="#">LD3 (multiple structures)</a> — <a href="#">register offset</a>
1	!= 11111	0110	<a href="#">LD1 (multiple structures)</a> — <a href="#">three registers, register offset</a>
1	!= 11111	0111	<a href="#">LD1 (multiple structures)</a> — <a href="#">one register, register offset</a>
1	!= 11111	1000	<a href="#">LD2 (multiple structures)</a> — <a href="#">register offset</a>
1	!= 11111	1010	<a href="#">LD1 (multiple structures)</a> — <a href="#">two registers, register offset</a>
1	11111	0000	<a href="#">LD4 (multiple structures)</a> — <a href="#">immediate offset</a>
1	11111	0010	<a href="#">LD1 (multiple structures)</a> — <a href="#">four registers, immediate offset</a>
1	11111	0100	<a href="#">LD3 (multiple structures)</a> — <a href="#">immediate offset</a>
1	11111	0110	<a href="#">LD1 (multiple structures)</a> — <a href="#">three registers, immediate offset</a>
1	11111	0111	<a href="#">LD1 (multiple structures)</a> — <a href="#">one register, immediate offset</a>
1	11111	1000	<a href="#">LD2 (multiple structures)</a> — <a href="#">immediate offset</a>
1	11111	1010	<a href="#">LD1 (multiple structures)</a> — <a href="#">two registers, immediate offset</a>

### Advanced SIMD load/store single structure

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	L	R	0	0	0	0	o2	opcode	S	size													

Decode fields						Instruction Details	Feature
L	R	o2	opcode	S	size		
	0	1	x00	0	00	UNALLOCATED	-
	0	1	x00	0	1x	UNALLOCATED	-
	0	1	x00	1		UNALLOCATED	-
	0	1	x01			UNALLOCATED	-
	0	1	x1x			UNALLOCATED	-
	1	1				UNALLOCATED	-
0		0	11x			UNALLOCATED	-



L	R	Decode fields		S	size	Instruction Details	Feature
		o2	opcode				
0	0	0	000			<a href="#">ST1 (single structure) — 8-bit</a>	-
0	0	0	001			<a href="#">ST3 (single structure) — 8-bit</a>	-
0	0	0	010		×0	<a href="#">ST1 (single structure) — 16-bit</a>	-
0	0	0	010		×1	UNALLOCATED	-
0	0	0	011		×0	<a href="#">ST3 (single structure) — 16-bit</a>	-
0	0	0	011		×1	UNALLOCATED	-
0	0	0	100		00	<a href="#">ST1 (single structure) — 32-bit</a>	-
0	0	0	100		1×	UNALLOCATED	-
0	0	0	100	0	01	<a href="#">ST1 (single structure) — 64-bit</a>	-
0	0	0	100	1	01	UNALLOCATED	-
0	0	0	101		00	<a href="#">ST3 (single structure) — 32-bit</a>	-
0	0	0	101		10	UNALLOCATED	-
0	0	0	101	0	01	<a href="#">ST3 (single structure) — 64-bit</a>	-
0	0	0	101	0	11	UNALLOCATED	-
0	0	0	101	1	×1	UNALLOCATED	-
0	0	1	100	0	01	<a href="#">STL1 (SIMD&amp;FP)</a>	FEAT_LRCPC3
0	1	0	000			<a href="#">ST2 (single structure) — 8-bit</a>	-
0	1	0	001			<a href="#">ST4 (single structure) — 8-bit</a>	-
0	1	0	010		×0	<a href="#">ST2 (single structure) — 16-bit</a>	-
0	1	0	010		×1	UNALLOCATED	-
0	1	0	011		×0	<a href="#">ST4 (single structure) — 16-bit</a>	-
0	1	0	011		×1	UNALLOCATED	-
0	1	0	100		00	<a href="#">ST2 (single structure) — 32-bit</a>	-
0	1	0	100		10	UNALLOCATED	-
0	1	0	100	0	01	<a href="#">ST2 (single structure) — 64-bit</a>	-
0	1	0	100	0	11	UNALLOCATED	-
0	1	0	100	1	×1	UNALLOCATED	-
0	1	0	101		00	<a href="#">ST4 (single structure) — 32-bit</a>	-
0	1	0	101		10	UNALLOCATED	-
0	1	0	101	0	01	<a href="#">ST4 (single structure) — 64-bit</a>	-
0	1	0	101	0	11	UNALLOCATED	-
0	1	0	101	1	×1	UNALLOCATED	-
1	0	0	000			<a href="#">LD1 (single structure) — 8-bit</a>	-
1	0	0	001			<a href="#">LD3 (single structure) — 8-bit</a>	-
1	0	0	010		×0	<a href="#">LD1 (single structure) — 16-bit</a>	-
1	0	0	010		×1	UNALLOCATED	-
1	0	0	011		×0	<a href="#">LD3 (single structure) — 16-bit</a>	-
1	0	0	011		×1	UNALLOCATED	-
1	0	0	100		00	<a href="#">LD1 (single structure) — 32-bit</a>	-
1	0	0	100		1×	UNALLOCATED	-
1	0	0	100	0	01	<a href="#">LD1 (single structure) — 64-bit</a>	-
1	0	0	100	1	01	UNALLOCATED	-
1	0	0	101		00	<a href="#">LD3 (single structure) — 32-bit</a>	-
1	0	0	101		10	UNALLOCATED	-
1	0	0	101	0	01	<a href="#">LD3 (single structure) — 64-bit</a>	-
1	0	0	101	0	11	UNALLOCATED	-
1	0	0	101	1	×1	UNALLOCATED	-

Decode fields						Instruction Details	Feature
L	R	o2	opcode	S	size		
1	0	0	110	0		<a href="#">LD1R</a>	-
1	0	0	110	1		UNALLOCATED	-
1	0	0	111	0		<a href="#">LD3R</a>	-
1	0	0	111	1		UNALLOCATED	-
1	0	1	100	0	01	<a href="#">LDAP1 (SIMD&amp;FP)</a>	FEAT_LRCPC3
1	1	0	000			<a href="#">LD2 (single structure) — 8-bit</a>	-
1	1	0	001			<a href="#">LD4 (single structure) — 8-bit</a>	-
1	1	0	010		×0	<a href="#">LD2 (single structure) — 16-bit</a>	-
1	1	0	010		×1	UNALLOCATED	-
1	1	0	011		×0	<a href="#">LD4 (single structure) — 16-bit</a>	-
1	1	0	011		×1	UNALLOCATED	-
1	1	0	100		00	<a href="#">LD2 (single structure) — 32-bit</a>	-
1	1	0	100		10	UNALLOCATED	-
1	1	0	100	0	01	<a href="#">LD2 (single structure) — 64-bit</a>	-
1	1	0	100	0	11	UNALLOCATED	-
1	1	0	100	1	×1	UNALLOCATED	-
1	1	0	101		00	<a href="#">LD4 (single structure) — 32-bit</a>	-
1	1	0	101		10	UNALLOCATED	-
1	1	0	101	0	01	<a href="#">LD4 (single structure) — 64-bit</a>	-
1	1	0	101	0	11	UNALLOCATED	-
1	1	0	101	1	×1	UNALLOCATED	-
1	1	0	110	0		<a href="#">LD2R</a>	-
1	1	0	110	1		UNALLOCATED	-
1	1	0	111	0		<a href="#">LD4R</a>	-
1	1	0	111	1		UNALLOCATED	-

**Advanced SIMD load/store single structure (post-indexed)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	L	R	Rm					opcode			S	size	Rn					Rt					

Decode fields						Instruction Details
L	R	Rm	opcode	S	size	
0			11×			UNALLOCATED
0	0		010		×1	UNALLOCATED
0	0		011		×1	UNALLOCATED
0	0		100		1×	UNALLOCATED
0	0		100	1	01	UNALLOCATED
0	0		101		10	UNALLOCATED
0	0		101	0	11	UNALLOCATED
0	0		101	1	×1	UNALLOCATED
0	0	!= 11111	000			<a href="#">ST1 (single structure) — 8-bit, register offset</a>
0	0	!= 11111	001			<a href="#">ST3 (single structure) — 8-bit, register offset</a>
0	0	!= 11111	010		×0	<a href="#">ST1 (single structure) — 16-bit, register offset</a>
0	0	!= 11111	011		×0	<a href="#">ST3 (single structure) — 16-bit, register offset</a>
0	0	!= 11111	100		00	<a href="#">ST1 (single structure) — 32-bit, register offset</a>
0	0	!= 11111	100	0	01	<a href="#">ST1 (single structure) — 64-bit, register offset</a>

L	R	Decode fields		S	size	Instruction Details
		Rm	opcode			
0	0	!= 11111	101		00	<a href="#">ST3 (single structure) — 32-bit, register offset</a>
0	0	!= 11111	101	0	01	<a href="#">ST3 (single structure) — 64-bit, register offset</a>
0	0	11111	000			<a href="#">ST1 (single structure) — 8-bit, immediate offset</a>
0	0	11111	001			<a href="#">ST3 (single structure) — 8-bit, immediate offset</a>
0	0	11111	010		×0	<a href="#">ST1 (single structure) — 16-bit, immediate offset</a>
0	0	11111	011		×0	<a href="#">ST3 (single structure) — 16-bit, immediate offset</a>
0	0	11111	100		00	<a href="#">ST1 (single structure) — 32-bit, immediate offset</a>
0	0	11111	100	0	01	<a href="#">ST1 (single structure) — 64-bit, immediate offset</a>
0	0	11111	101		00	<a href="#">ST3 (single structure) — 32-bit, immediate offset</a>
0	0	11111	101	0	01	<a href="#">ST3 (single structure) — 64-bit, immediate offset</a>
0	1		010		×1	UNALLOCATED
0	1		011		×1	UNALLOCATED
0	1		100		10	UNALLOCATED
0	1		100	0	11	UNALLOCATED
0	1		100	1	×1	UNALLOCATED
0	1		101		10	UNALLOCATED
0	1		101	0	11	UNALLOCATED
0	1		101	1	×1	UNALLOCATED
0	1	!= 11111	000			<a href="#">ST2 (single structure) — 8-bit, register offset</a>
0	1	!= 11111	001			<a href="#">ST4 (single structure) — 8-bit, register offset</a>
0	1	!= 11111	010		×0	<a href="#">ST2 (single structure) — 16-bit, register offset</a>
0	1	!= 11111	011		×0	<a href="#">ST4 (single structure) — 16-bit, register offset</a>
0	1	!= 11111	100		00	<a href="#">ST2 (single structure) — 32-bit, register offset</a>
0	1	!= 11111	100	0	01	<a href="#">ST2 (single structure) — 64-bit, register offset</a>
0	1	!= 11111	101		00	<a href="#">ST4 (single structure) — 32-bit, register offset</a>
0	1	!= 11111	101	0	01	<a href="#">ST4 (single structure) — 64-bit, register offset</a>
0	1	11111	000			<a href="#">ST2 (single structure) — 8-bit, immediate offset</a>
0	1	11111	001			<a href="#">ST4 (single structure) — 8-bit, immediate offset</a>
0	1	11111	010		×0	<a href="#">ST2 (single structure) — 16-bit, immediate offset</a>
0	1	11111	011		×0	<a href="#">ST4 (single structure) — 16-bit, immediate offset</a>
0	1	11111	100		00	<a href="#">ST2 (single structure) — 32-bit, immediate offset</a>
0	1	11111	100	0	01	<a href="#">ST2 (single structure) — 64-bit, immediate offset</a>
0	1	11111	101		00	<a href="#">ST4 (single structure) — 32-bit, immediate offset</a>
0	1	11111	101	0	01	<a href="#">ST4 (single structure) — 64-bit, immediate offset</a>
1	0		010		×1	UNALLOCATED
1	0		011		×1	UNALLOCATED
1	0		100		1×	UNALLOCATED
1	0		100	1	01	UNALLOCATED
1	0		101		10	UNALLOCATED
1	0		101	0	11	UNALLOCATED
1	0		101	1	×1	UNALLOCATED
1	0		110	1		UNALLOCATED
1	0		111	1		UNALLOCATED
1	0	!= 11111	000			<a href="#">LD1 (single structure) — 8-bit, register offset</a>
1	0	!= 11111	001			<a href="#">LD3 (single structure) — 8-bit, register offset</a>
1	0	!= 11111	010		×0	<a href="#">LD1 (single structure) — 16-bit, register offset</a>
1	0	!= 11111	011		×0	<a href="#">LD3 (single structure) — 16-bit, register offset</a>

L	R	Decode fields		S	size	Instruction Details
		Rm	opcode			
1	0	!= 11111	100		00	<a href="#">LD1 (single structure) — 32-bit, register offset</a>
1	0	!= 11111	100	0	01	<a href="#">LD1 (single structure) — 64-bit, register offset</a>
1	0	!= 11111	101		00	<a href="#">LD3 (single structure) — 32-bit, register offset</a>
1	0	!= 11111	101	0	01	<a href="#">LD3 (single structure) — 64-bit, register offset</a>
1	0	!= 11111	110	0		<a href="#">LD1R — register offset</a>
1	0	!= 11111	111	0		<a href="#">LD3R — register offset</a>
1	0	11111	000			<a href="#">LD1 (single structure) — 8-bit, immediate offset</a>
1	0	11111	001			<a href="#">LD3 (single structure) — 8-bit, immediate offset</a>
1	0	11111	010		×0	<a href="#">LD1 (single structure) — 16-bit, immediate offset</a>
1	0	11111	011		×0	<a href="#">LD3 (single structure) — 16-bit, immediate offset</a>
1	0	11111	100		00	<a href="#">LD1 (single structure) — 32-bit, immediate offset</a>
1	0	11111	100	0	01	<a href="#">LD1 (single structure) — 64-bit, immediate offset</a>
1	0	11111	101		00	<a href="#">LD3 (single structure) — 32-bit, immediate offset</a>
1	0	11111	101	0	01	<a href="#">LD3 (single structure) — 64-bit, immediate offset</a>
1	0	11111	110	0		<a href="#">LD1R — immediate offset</a>
1	0	11111	111	0		<a href="#">LD3R — immediate offset</a>
1	1		010		×1	UNALLOCATED
1	1		011		×1	UNALLOCATED
1	1		100		10	UNALLOCATED
1	1		100	0	11	UNALLOCATED
1	1		100	1	×1	UNALLOCATED
1	1		101		10	UNALLOCATED
1	1		101	0	11	UNALLOCATED
1	1		101	1	×1	UNALLOCATED
1	1		110	1		UNALLOCATED
1	1		111	1		UNALLOCATED
1	1	!= 11111	000			<a href="#">LD2 (single structure) — 8-bit, register offset</a>
1	1	!= 11111	001			<a href="#">LD4 (single structure) — 8-bit, register offset</a>
1	1	!= 11111	010		×0	<a href="#">LD2 (single structure) — 16-bit, register offset</a>
1	1	!= 11111	011		×0	<a href="#">LD4 (single structure) — 16-bit, register offset</a>
1	1	!= 11111	100		00	<a href="#">LD2 (single structure) — 32-bit, register offset</a>
1	1	!= 11111	100	0	01	<a href="#">LD2 (single structure) — 64-bit, register offset</a>
1	1	!= 11111	101		00	<a href="#">LD4 (single structure) — 32-bit, register offset</a>
1	1	!= 11111	101	0	01	<a href="#">LD4 (single structure) — 64-bit, register offset</a>
1	1	!= 11111	110	0		<a href="#">LD2R — register offset</a>
1	1	!= 11111	111	0		<a href="#">LD4R — register offset</a>
1	1	11111	000			<a href="#">LD2 (single structure) — 8-bit, immediate offset</a>
1	1	11111	001			<a href="#">LD4 (single structure) — 8-bit, immediate offset</a>
1	1	11111	010		×0	<a href="#">LD2 (single structure) — 16-bit, immediate offset</a>
1	1	11111	011		×0	<a href="#">LD4 (single structure) — 16-bit, immediate offset</a>
1	1	11111	100		00	<a href="#">LD2 (single structure) — 32-bit, immediate offset</a>
1	1	11111	100	0	01	<a href="#">LD2 (single structure) — 64-bit, immediate offset</a>
1	1	11111	101		00	<a href="#">LD4 (single structure) — 32-bit, immediate offset</a>
1	1	11111	101	0	01	<a href="#">LD4 (single structure) — 64-bit, immediate offset</a>
1	1	11111	110	0		<a href="#">LD2R — immediate offset</a>
1	1	11111	111	0		<a href="#">LD4R — immediate offset</a>

**RCW compare and swap**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	S	0	1	1	0	0	1	A	R	1					Rs		0	0	0	0	1	0									Rt

Decode fields			Instruction Details	Feature
S	A	R		
0	0	0	<a href="#">RCWCAS, RCWCASA, RCWCASL, RCWCASAL</a> — <a href="#">RCWCAS</a>	FEAT_THE
0	0	1	<a href="#">RCWCAS, RCWCASA, RCWCASL, RCWCASAL</a> — <a href="#">RCWCASL</a>	FEAT_THE
0	1	0	<a href="#">RCWCAS, RCWCASA, RCWCASL, RCWCASAL</a> — <a href="#">RCWCASA</a>	FEAT_THE
0	1	1	<a href="#">RCWCAS, RCWCASA, RCWCASL, RCWCASAL</a> — <a href="#">RCWCASAL</a>	FEAT_THE
1	0	0	<a href="#">RCWSCAS, RCWSCASA, RCWSCASL, RCWSCASAL</a> — <a href="#">RCWSCAS</a>	FEAT_THE
1	0	1	<a href="#">RCWSCAS, RCWSCASA, RCWSCASL, RCWSCASAL</a> — <a href="#">RCWSCASL</a>	FEAT_THE
1	1	0	<a href="#">RCWSCAS, RCWSCASA, RCWSCASL, RCWSCASAL</a> — <a href="#">RCWSCASA</a>	FEAT_THE
1	1	1	<a href="#">RCWSCAS, RCWSCASA, RCWSCASL, RCWSCASAL</a> — <a href="#">RCWSCASAL</a>	FEAT_THE

**RCW compare and swap pair**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	S	0	1	1	0	0	1	A	R	1					Rs		0	0	0	0	1	1									Rt

Decode fields			Instruction Details	Feature
S	A	R		
0	0	0	<a href="#">RCWCASP, RCWCASPA, RCWCASPL, RCWCASPAL</a> — <a href="#">RCWCASP</a>	FEAT_D128 && FEAT_THE
0	0	1	<a href="#">RCWCASP, RCWCASPA, RCWCASPL, RCWCASPAL</a> — <a href="#">RCWCASPL</a>	FEAT_D128 && FEAT_THE
0	1	0	<a href="#">RCWCASP, RCWCASPA, RCWCASPL, RCWCASPAL</a> — <a href="#">RCWCASPA</a>	FEAT_D128 && FEAT_THE
0	1	1	<a href="#">RCWCASP, RCWCASPA, RCWCASPL, RCWCASPAL</a> — <a href="#">RCWCASPAL</a>	FEAT_D128 && FEAT_THE
1	0	0	<a href="#">RCWSCASP, RCWSCASPA, RCWSCASPL, RCWSCASPAL</a> — <a href="#">RCWSCASP</a>	FEAT_D128 && FEAT_THE
1	0	1	<a href="#">RCWSCASP, RCWSCASPA, RCWSCASPL, RCWSCASPAL</a> — <a href="#">RCWSCASPL</a>	FEAT_D128 && FEAT_THE
1	1	0	<a href="#">RCWSCASP, RCWSCASPA, RCWSCASPL, RCWSCASPAL</a> — <a href="#">RCWSCASPA</a>	FEAT_D128 && FEAT_THE
1	1	1	<a href="#">RCWSCASP, RCWSCASPA, RCWSCASPL, RCWSCASPAL</a> — <a href="#">RCWSCASPAL</a>	FEAT_D128 && FEAT_THE

**128-bit atomic memory operations**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	S	0	1	1	0	0	1	A	R	1					Rt2		o3		opc		0	0									Rt

Decode fields					Instruction Details	Feature
S	A	R	o3	opc		
0				1xx	UNALLOCATED	-
0			0	0x0	UNALLOCATED	-
0	0	0	0	001	<a href="#">LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL</a> — <a href="#">LDCLRP</a>	FEAT_LSE128
0	0	0	0	011	<a href="#">LDSETP, LDSETPA, LDSETPAL, LDSETPL</a> — <a href="#">LDSETP</a>	FEAT_LSE128

Decode fields					Instruction Details	Feature
S	A	R	o3	opc		
0	0	0	1	000	<a href="#">SWPP, SWPPA, SWPPAL, SWPPL — SWPP</a>	FEAT_LSE128
0	0	0	1	001	<a href="#">RCWCLRP, RCWCLRPA, RCWCLRPL, RCWCLRPAL — RCWCLRP</a>	FEAT_D128 && FEAT_THE
0	0	0	1	010	<a href="#">RCWSWPP, RCWSWPPA, RCWSWPPL, RCWSWPPAL — RCWSWPP</a>	FEAT_D128 && FEAT_THE
0	0	0	1	011	<a href="#">RCWSETP, RCWSETPA, RCWSETPL, RCWSETPAL — RCWSETP</a>	FEAT_D128 && FEAT_THE
0	0	1	0	001	<a href="#">LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL — LDCLRPL</a>	FEAT_LSE128
0	0	1	0	011	<a href="#">LDSETP, LDSETPA, LDSETPAL, LDSETPL — LDSETPL</a>	FEAT_LSE128
0	0	1	1	000	<a href="#">SWPP, SWPPA, SWPPAL, SWPPL — SWPPL</a>	FEAT_LSE128
0	0	1	1	001	<a href="#">RCWCLRP, RCWCLRPA, RCWCLRPL, RCWCLRPAL — RCWCLRPL</a>	FEAT_D128 && FEAT_THE
0	0	1	1	010	<a href="#">RCWSWPP, RCWSWPPA, RCWSWPPL, RCWSWPPAL — RCWSWPPL</a>	FEAT_D128 && FEAT_THE
0	0	1	1	011	<a href="#">RCWSETP, RCWSETPA, RCWSETPL, RCWSETPAL — RCWSETPL</a>	FEAT_D128 && FEAT_THE
0	1	0	0	001	<a href="#">LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL — LDCLRPA</a>	FEAT_LSE128
0	1	0	0	011	<a href="#">LDSETP, LDSETPA, LDSETPAL, LDSETPL — LDSETPA</a>	FEAT_LSE128
0	1	0	1	000	<a href="#">SWPP, SWPPA, SWPPAL, SWPPL — SWPPA</a>	FEAT_LSE128
0	1	0	1	001	<a href="#">RCWCLRP, RCWCLRPA, RCWCLRPL, RCWCLRPAL — RCWCLRPA</a>	FEAT_D128 && FEAT_THE
0	1	0	1	010	<a href="#">RCWSWPP, RCWSWPPA, RCWSWPPL, RCWSWPPAL — RCWSWPPA</a>	FEAT_D128 && FEAT_THE
0	1	0	1	011	<a href="#">RCWSETP, RCWSETPA, RCWSETPL, RCWSETPAL — RCWSETPA</a>	FEAT_D128 && FEAT_THE
0	1	1	0	001	<a href="#">LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL — LDCLRPAL</a>	FEAT_LSE128
0	1	1	0	011	<a href="#">LDSETP, LDSETPA, LDSETPAL, LDSETPL — LDSETPAL</a>	FEAT_LSE128
0	1	1	1	000	<a href="#">SWPP, SWPPA, SWPPAL, SWPPL — SWPPAL</a>	FEAT_LSE128
0	1	1	1	001	<a href="#">RCWCLRP, RCWCLRPA, RCWCLRPL, RCWCLRPAL — RCWCLRPAL</a>	FEAT_D128 && FEAT_THE
0	1	1	1	010	<a href="#">RCWSWPP, RCWSWPPA, RCWSWPPL, RCWSWPPAL — RCWSWPPAL</a>	FEAT_D128 && FEAT_THE
0	1	1	1	011	<a href="#">RCWSETP, RCWSETPA, RCWSETPL, RCWSETPAL — RCWSETPAL</a>	FEAT_D128 && FEAT_THE
1			0		UNALLOCATED	-
1			1	000	UNALLOCATED	-
1			1	1xx	UNALLOCATED	-
1	0	0	1	001	<a href="#">RCWSCLRP, RCWSCLRPA, RCWSCLRPL, RCWSCLRPAL — RCWSCLRP</a>	FEAT_D128 && FEAT_THE
1	0	0	1	010	<a href="#">RCWSSWPP, RCWSSWPPA, RCWSSWPPL, RCWSSWPPAL — RCWSSWPP</a>	FEAT_D128 && FEAT_THE
1	0	0	1	011	<a href="#">RCWSSETP, RCWSSETPA, RCWSSETPL, RCWSSETPAL — RCWSSETP</a>	FEAT_D128 && FEAT_THE
1	0	1	1	001	<a href="#">RCWSCLRP, RCWSCLRPA, RCWSCLRPL, RCWSCLRPAL — RCWSCLRPL</a>	FEAT_D128 && FEAT_THE
1	0	1	1	010	<a href="#">RCWSSWPP, RCWSSWPPA, RCWSSWPPL, RCWSSWPPAL — RCWSSWPPL</a>	FEAT_D128 && FEAT_THE
1	0	1	1	011	<a href="#">RCWSSETP, RCWSSETPA, RCWSSETPL, RCWSSETPAL — RCWSSETPL</a>	FEAT_D128 && FEAT_THE
1	1	0	1	001	<a href="#">RCWSCLRP, RCWSCLRPA, RCWSCLRPL, RCWSCLRPAL — RCWSCLRPAL</a>	FEAT_D128 && FEAT_THE
1	1	0	1	010	<a href="#">RCWSSWPP, RCWSSWPPA, RCWSSWPPL, RCWSSWPPAL — RCWSSWPPA</a>	FEAT_D128 && FEAT_THE
1	1	0	1	011	<a href="#">RCWSSETP, RCWSSETPA, RCWSSETPL, RCWSSETPAL — RCWSSETPA</a>	FEAT_D128 && FEAT_THE

Decode fields					Instruction Details	Feature
S	A	R	o3	opc		
1	1	1	1	001	<a href="#">RCWSCLR, RCWSCLRPA, RCWSCLRPL, RCWSCLRPA — RCWSCLRPA</a>	FEAT_D128 && FEAT_THE
1	1	1	1	010	<a href="#">RCWSSWPP, RCWSSWPPA, RCWSSWPPL, RCWSSWPPAL — RCWSSWPPAL</a>	FEAT_D128 && FEAT_THE
1	1	1	1	011	<a href="#">RCWSSETP, RCWSSETPA, RCWSSETPL, RCWSSETPAL — RCWSSETPAL</a>	FEAT_D128 && FEAT_THE

### GCS load/store

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	0	0	0	1	1	1	1	1	0	opc			1	1	Rn				Rt					

Decode fields opc	Instruction Details	Feature
000	<a href="#">GCSSTR</a>	FEAT_GCS
001	<a href="#">GCSSTTR</a>	FEAT_GCS
01x	UNALLOCATED	-
1xx	UNALLOCATED	-

### Load/store memory tags

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	opc		1	imm9								op2		Rn				Rt						

opc	Decode fields imm9	op2	Instruction Details	Feature
00		01	<a href="#">STG</a> — <a href="#">post-index</a>	FEAT_MTE
00		10	<a href="#">STG</a> — <a href="#">signed offset</a>	FEAT_MTE
00		11	<a href="#">STG</a> — <a href="#">pre-index</a>	FEAT_MTE
00	000000000	00	<a href="#">STZGM</a>	FEAT_MTE2
01		00	<a href="#">LDG</a>	FEAT_MTE
01		01	<a href="#">STZG</a> — <a href="#">post-index</a>	FEAT_MTE
01		10	<a href="#">STZG</a> — <a href="#">signed offset</a>	FEAT_MTE
01		11	<a href="#">STZG</a> — <a href="#">pre-index</a>	FEAT_MTE
10		01	<a href="#">ST2G</a> — <a href="#">post-index</a>	FEAT_MTE
10		10	<a href="#">ST2G</a> — <a href="#">signed offset</a>	FEAT_MTE
10		11	<a href="#">ST2G</a> — <a href="#">pre-index</a>	FEAT_MTE
10	!= 000000000	00	UNALLOCATED	-
10	000000000	00	<a href="#">STGM</a>	FEAT_MTE2
11		01	<a href="#">STZ2G</a> — <a href="#">post-index</a>	FEAT_MTE
11		10	<a href="#">STZ2G</a> — <a href="#">signed offset</a>	FEAT_MTE
11		11	<a href="#">STZ2G</a> — <a href="#">pre-index</a>	FEAT_MTE
11	!= 000000000	00	UNALLOCATED	-
11	000000000	00	<a href="#">LDGM</a>	FEAT_MTE2

### Load/store exclusive pair

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	sz	0	0	1	0	0	0	0	L	1			Rs			o0			Rt2					Rn					Rt		

Decode fields			Instruction Details
sz	L	o0	
0	0	0	<a href="#">STXP — 32-bit</a>
0	0	1	<a href="#">STLXP — 32-bit</a>
0	1	0	<a href="#">LDXP — 32-bit</a>
0	1	1	<a href="#">LDAXP — 32-bit</a>
1	0	0	<a href="#">STXP — 64-bit</a>
1	0	1	<a href="#">STLXP — 64-bit</a>
1	1	0	<a href="#">LDXP — 64-bit</a>
1	1	1	<a href="#">LDAXP — 64-bit</a>

**Load/store exclusive register**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	0	0	1	0	0	0	0	0	L	0			Rs			o0			Rt2					Rn					Rt		

Decode fields			Instruction Details
size	L	o0	
00	0	0	<a href="#">STXRB</a>
00	0	1	<a href="#">STLXRB</a>
00	1	0	<a href="#">LDXRB</a>
00	1	1	<a href="#">LDAXRB</a>
01	0	0	<a href="#">STXRH</a>
01	0	1	<a href="#">STLXRH</a>
01	1	0	<a href="#">LDXRH</a>
01	1	1	<a href="#">LDAXRH</a>
10	0	0	<a href="#">STXR — 32-bit</a>
10	0	1	<a href="#">STLXR — 32-bit</a>
10	1	0	<a href="#">LDXR — 32-bit</a>
10	1	1	<a href="#">LDAXR — 32-bit</a>
11	0	0	<a href="#">STXR — 64-bit</a>
11	0	1	<a href="#">STLXR — 64-bit</a>
11	1	0	<a href="#">LDXR — 64-bit</a>
11	1	1	<a href="#">LDAXR — 64-bit</a>

**Load/store ordered**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	0	0	1	0	0	0	1	1	L	0			Rs			o0			Rt2					Rn					Rt		

Decode fields			Instruction Details	Feature
size	L	o0		
00	0	0	<a href="#">STLLRB</a>	FEAT_LOR
00	0	1	<a href="#">STLRB</a>	-
00	1	0	<a href="#">LDLARB</a>	FEAT_LOR
00	1	1	<a href="#">LDARB</a>	-
01	0	0	<a href="#">STLLRH</a>	FEAT_LOR



Decode fields size	L	o0	Instruction Details	Feature
01	0	1	<a href="#">STLRH</a>	-
01	1	0	<a href="#">LDLARH</a>	FEAT_LOR
01	1	1	<a href="#">LDARH</a>	-
10	0	0	<a href="#">STLLR — 32-bit</a>	FEAT_LOR
10	0	1	<a href="#">STLR — 32-bit</a>	-
10	1	0	<a href="#">LDLAR — 32-bit</a>	FEAT_LOR
10	1	1	<a href="#">LDAR — 32-bit</a>	-
11	0	0	<a href="#">STLLR — 64-bit</a>	FEAT_LOR
11	0	1	<a href="#">STLR — 64-bit</a>	-
11	1	0	<a href="#">LDLAR — 64-bit</a>	FEAT_LOR
11	1	1	<a href="#">LDAR — 64-bit</a>	-

## Compare and swap

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	0	0	1	0	0	0	1	L	1				Rs			o0			Rt2				Rn						Rt		

Decode fields size	L	o0	Rt2	Instruction Details	Feature
			!= 11111	UNALLOCATED	-
00	0	0	11111	<a href="#">CASB, CASAB, CASALB, CASLB — CASB</a>	FEAT_LSE
00	0	1	11111	<a href="#">CASB, CASAB, CASALB, CASLB — CASLB</a>	FEAT_LSE
00	1	0	11111	<a href="#">CASB, CASAB, CASALB, CASLB — CASAB</a>	FEAT_LSE
00	1	1	11111	<a href="#">CASB, CASAB, CASALB, CASLB — CASALB</a>	FEAT_LSE
01	0	0	11111	<a href="#">CASH, CASAH, CASALH, CASLH — CASH</a>	FEAT_LSE
01	0	1	11111	<a href="#">CASH, CASAH, CASALH, CASLH — CASLH</a>	FEAT_LSE
01	1	0	11111	<a href="#">CASH, CASAH, CASALH, CASLH — CASAH</a>	FEAT_LSE
01	1	1	11111	<a href="#">CASH, CASAH, CASALH, CASLH — CASALH</a>	FEAT_LSE
10	0	0	11111	<a href="#">CAS, CASA, CASAL, CASL — 32-bit CAS</a>	FEAT_LSE
10	0	1	11111	<a href="#">CAS, CASA, CASAL, CASL — 32-bit CASL</a>	FEAT_LSE
10	1	0	11111	<a href="#">CAS, CASA, CASAL, CASL — 32-bit CASA</a>	FEAT_LSE
10	1	1	11111	<a href="#">CAS, CASA, CASAL, CASL — 32-bit CASAL</a>	FEAT_LSE
11	0	0	11111	<a href="#">CAS, CASA, CASAL, CASL — 64-bit CAS</a>	FEAT_LSE
11	0	1	11111	<a href="#">CAS, CASA, CASAL, CASL — 64-bit CASL</a>	FEAT_LSE
11	1	0	11111	<a href="#">CAS, CASA, CASAL, CASL — 64-bit CASA</a>	FEAT_LSE
11	1	1	11111	<a href="#">CAS, CASA, CASAL, CASL — 64-bit CASAL</a>	FEAT_LSE

## LDIAPP/STILP

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	0	1	1	0	0	1	0	L	0						Rt2			opc2		1	0			Rn					Rt		

Decode fields size	L	opc2	Instruction Details	Feature
0x			UNALLOCATED	-
1x		001x	UNALLOCATED	-

Decode fields size	L	opc2	Instruction Details	Feature
1x		01xx	UNALLOCATED	-
1x		1xxx	UNALLOCATED	-
10	0	0000	<a href="#">STILP — 32-bit pre-index</a>	FEAT_LRCPC3
10	0	0001	<a href="#">STILP — 32-bit</a>	FEAT_LRCPC3
10	1	0000	<a href="#">LDIAPP — 32-bit post-index</a>	FEAT_LRCPC3
10	1	0001	<a href="#">LDIAPP — 32-bit</a>	FEAT_LRCPC3
11	0	0000	<a href="#">STILP — 64-bit pre-index</a>	FEAT_LRCPC3
11	0	0001	<a href="#">STILP — 64-bit</a>	FEAT_LRCPC3
11	1	0000	<a href="#">LDIAPP — 64-bit post-index</a>	FEAT_LRCPC3
11	1	0001	<a href="#">LDIAPP — 64-bit</a>	FEAT_LRCPC3

**LDAPR/STLR (writeback)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	0	1	1	0	0	1	1	L	0	0	0	0	0	0	0	0	0	0	0	1	0	Rn				Rt					

Decode fields size	L	Instruction Details	Feature
0x		UNALLOCATED	-
10	0	<a href="#">STLR — 32-bit</a>	FEAT_LRCPC3
10	1	<a href="#">LDAPR — 32-bit</a>	FEAT_LRCPC3
11	0	<a href="#">STLR — 64-bit</a>	FEAT_LRCPC3
11	1	<a href="#">LDAPR — 64-bit</a>	FEAT_LRCPC3

**LDAPR/STLR (unscaled immediate)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
size		0	1	1	0	0	1	opc		0	imm9										0	0	Rn					Rt				

Decode fields size	opc	Instruction Details	Feature
00	00	<a href="#">STLURB</a>	FEAT_LRCPC2
00	01	<a href="#">LDAPURB</a>	FEAT_LRCPC2
00	10	<a href="#">LDAPURSB — 64-bit</a>	FEAT_LRCPC2
00	11	<a href="#">LDAPURSB — 32-bit</a>	FEAT_LRCPC2
01	00	<a href="#">STLURH</a>	FEAT_LRCPC2
01	01	<a href="#">LDAPURH</a>	FEAT_LRCPC2
01	10	<a href="#">LDAPURSH — 64-bit</a>	FEAT_LRCPC2
01	11	<a href="#">LDAPURSH — 32-bit</a>	FEAT_LRCPC2
1x	11	UNALLOCATED	-
10	00	<a href="#">STLUR — 32-bit</a>	FEAT_LRCPC2
10	01	<a href="#">LDAPUR — 32-bit</a>	FEAT_LRCPC2
10	10	<a href="#">LDAPURSW</a>	FEAT_LRCPC2
11	00	<a href="#">STLUR — 64-bit</a>	FEAT_LRCPC2
11	01	<a href="#">LDAPUR — 64-bit</a>	FEAT_LRCPC2
11	10	UNALLOCATED	-

**LDAPR/STLR (SIMD&FP)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	0	1	1	1	0	1		opc	0												1	0									Rt

Decode fields size	opc	Instruction Details	Feature
00	00	<a href="#">STLUR (SIMD&amp;FP) — 8-bit</a>	FEAT_LRCPC3
00	01	<a href="#">LDAPUR (SIMD&amp;FP) — 8-bit</a>	FEAT_LRCPC3
00	10	<a href="#">STLUR (SIMD&amp;FP) — 128-bit</a>	FEAT_LRCPC3
00	11	<a href="#">LDAPUR (SIMD&amp;FP) — 128-bit</a>	FEAT_LRCPC3
01	00	<a href="#">STLUR (SIMD&amp;FP) — 16-bit</a>	FEAT_LRCPC3
01	01	<a href="#">LDAPUR (SIMD&amp;FP) — 16-bit</a>	FEAT_LRCPC3
01	1x	UNALLOCATED	-
1x	1x	UNALLOCATED	-
10	00	<a href="#">STLUR (SIMD&amp;FP) — 32-bit</a>	FEAT_LRCPC3
10	01	<a href="#">LDAPUR (SIMD&amp;FP) — 32-bit</a>	FEAT_LRCPC3
11	00	<a href="#">STLUR (SIMD&amp;FP) — 64-bit</a>	FEAT_LRCPC3
11	01	<a href="#">LDAPUR (SIMD&amp;FP) — 64-bit</a>	FEAT_LRCPC3

**Load register (literal)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc	0	1	1	VR	0	0																									Rt

Decode fields opc	VR	Instruction Details
00	0	<a href="#">LDR (literal) — 32-bit</a>
00	1	<a href="#">LDR (literal, SIMD&amp;FP) — 32-bit</a>
01	0	<a href="#">LDR (literal) — 64-bit</a>
01	1	<a href="#">LDR (literal, SIMD&amp;FP) — 64-bit</a>
10	0	<a href="#">LDRSW (literal)</a>
10	1	<a href="#">LDR (literal, SIMD&amp;FP) — 128-bit</a>
11	0	<a href="#">PRFM (literal)</a>
11	1	UNALLOCATED

**Memory Copy and Memory Set**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	0	1	1	o0	0	1		op1	0																						Rd

Decode fields o0	op1	op2	Instruction Details	Feature
0	00	0000	<a href="#">CPYFP, CPYFM, CPYFE — CPYFP</a>	FEAT_MOPS
0	00	0001	<a href="#">CPYFPWT, CPYFMWT, CPYFEWT — CPYFPWT</a>	FEAT_MOPS
0	00	0010	<a href="#">CPYFPRT, CPYFMRT, CPYFERT — CPYFPRT</a>	FEAT_MOPS
0	00	0011	<a href="#">CPYFPT, CPYFMT, CPYFET — CPYFPT</a>	FEAT_MOPS
0	00	0100	<a href="#">CPYFPWN, CPYFMWN, CPYFEWN — CPYFPWN</a>	FEAT_MOPS
0	00	0101	<a href="#">CPYFPWTWN, CPYFMWTWN, CPYFEWTWN — CPYFPWTWN</a>	FEAT_MOPS

Decode fields			Instruction Details	Feature
o0	op1	op2		
0	00	0110	<a href="#">CPYFPRTWN, CPYFMRTWN, CPYFERTWN — CPYFPRTWN</a>	FEAT_MOPS
0	00	0111	<a href="#">CPYFPTWN, CPYFMTWN, CPYFETWN — CPYFPTWN</a>	FEAT_MOPS
0	00	1000	<a href="#">CPYFPRN, CPYFMRN, CPYFERN — CPYFPRN</a>	FEAT_MOPS
0	00	1001	<a href="#">CPYFPWTRN, CPYFMWTRN, CPYFEWTRN — CPYFPWTRN</a>	FEAT_MOPS
0	00	1010	<a href="#">CPYFPRTRN, CPYFMRTRN, CPYFERTRN — CPYFPRTRN</a>	FEAT_MOPS
0	00	1011	<a href="#">CPYFPTRN, CPYFMTRN, CPYFETRN — CPYFPTRN</a>	FEAT_MOPS
0	00	1100	<a href="#">CPYFPN, CPYFMN, CPYFEN — CPYFPN</a>	FEAT_MOPS
0	00	1101	<a href="#">CPYFPWTN, CPYFMWTN, CPYFEWTN — CPYFPWTN</a>	FEAT_MOPS
0	00	1110	<a href="#">CPYFPRTN, CPYFMRTN, CPYFERTN — CPYFPRTN</a>	FEAT_MOPS
0	00	1111	<a href="#">CPYFPTN, CPYFMTN, CPYFETN — CPYFPTN</a>	FEAT_MOPS
0	01	0000	<a href="#">CPYFP, CPYFM, CPYFE — CPYFM</a>	FEAT_MOPS
0	01	0001	<a href="#">CPYFPWT, CPYFMWT, CPYFEWT — CPYFMWT</a>	FEAT_MOPS
0	01	0010	<a href="#">CPYFPRT, CPYFMRT, CPYFERT — CPYFMRT</a>	FEAT_MOPS
0	01	0011	<a href="#">CPYFPT, CPYFMT, CPYFET — CPYFMT</a>	FEAT_MOPS
0	01	0100	<a href="#">CPYFPWN, CPYFMWN, CPYFEWN — CPYFMWN</a>	FEAT_MOPS
0	01	0101	<a href="#">CPYFPWTWN, CPYFMWTWN, CPYFEWTWN — CPYFMWTWN</a>	FEAT_MOPS
0	01	0110	<a href="#">CPYFPRTWN, CPYFMRTWN, CPYFERTWN — CPYFMRTWN</a>	FEAT_MOPS
0	01	0111	<a href="#">CPYFPTWN, CPYFMTWN, CPYFETWN — CPYFMTWN</a>	FEAT_MOPS
0	01	1000	<a href="#">CPYFPRN, CPYFMRN, CPYFERN — CPYFMRN</a>	FEAT_MOPS
0	01	1001	<a href="#">CPYFPWTRN, CPYFMWTRN, CPYFEWTRN — CPYFMWTRN</a>	FEAT_MOPS
0	01	1010	<a href="#">CPYFPRTRN, CPYFMRTRN, CPYFERTRN — CPYFMRTRN</a>	FEAT_MOPS
0	01	1011	<a href="#">CPYFPTRN, CPYFMTRN, CPYFETRN — CPYFMTRN</a>	FEAT_MOPS
0	01	1100	<a href="#">CPYFPN, CPYFMN, CPYFEN — CPYFMN</a>	FEAT_MOPS
0	01	1101	<a href="#">CPYFPWTN, CPYFMWTN, CPYFEWTN — CPYFMWTN</a>	FEAT_MOPS
0	01	1110	<a href="#">CPYFPRTN, CPYFMRTN, CPYFERTN — CPYFMRTN</a>	FEAT_MOPS
0	01	1111	<a href="#">CPYFPTN, CPYFMTN, CPYFETN — CPYFMTN</a>	FEAT_MOPS
0	10	0000	<a href="#">CPYFP, CPYFM, CPYFE — CPYFE</a>	FEAT_MOPS
0	10	0001	<a href="#">CPYFPWT, CPYFMWT, CPYFEWT — CPYFEWT</a>	FEAT_MOPS
0	10	0010	<a href="#">CPYFPRT, CPYFMRT, CPYFERT — CPYFERT</a>	FEAT_MOPS
0	10	0011	<a href="#">CPYFPT, CPYFMT, CPYFET — CPYFET</a>	FEAT_MOPS
0	10	0100	<a href="#">CPYFPWN, CPYFMWN, CPYFEWN — CPYFEWN</a>	FEAT_MOPS
0	10	0101	<a href="#">CPYFPWTWN, CPYFMWTWN, CPYFEWTWN — CPYFEWTWN</a>	FEAT_MOPS
0	10	0110	<a href="#">CPYFPRTWN, CPYFMRTWN, CPYFERTWN — CPYFERTWN</a>	FEAT_MOPS
0	10	0111	<a href="#">CPYFPTWN, CPYFMTWN, CPYFETWN — CPYFETWN</a>	FEAT_MOPS
0	10	1000	<a href="#">CPYFPRN, CPYFMRN, CPYFERN — CPYFERN</a>	FEAT_MOPS
0	10	1001	<a href="#">CPYFPWTRN, CPYFMWTRN, CPYFEWTRN — CPYFEWTRN</a>	FEAT_MOPS
0	10	1010	<a href="#">CPYFPRTRN, CPYFMRTRN, CPYFERTRN — CPYFERTRN</a>	FEAT_MOPS
0	10	1011	<a href="#">CPYFPTRN, CPYFMTRN, CPYFETRN — CPYFETRN</a>	FEAT_MOPS
0	10	1100	<a href="#">CPYFPN, CPYFMN, CPYFEN — CPYFEN</a>	FEAT_MOPS
0	10	1101	<a href="#">CPYFPWTN, CPYFMWTN, CPYFEWTN — CPYFEWTN</a>	FEAT_MOPS
0	10	1110	<a href="#">CPYFPRTN, CPYFMRTN, CPYFERTN — CPYFERTN</a>	FEAT_MOPS
0	10	1111	<a href="#">CPYFPTN, CPYFMTN, CPYFETN — CPYFETN</a>	FEAT_MOPS
0	11	0000	<a href="#">SETP, SETM, SETE — SETP</a>	FEAT_MOPS
0	11	0001	<a href="#">SETPT, SETMT, SETET — SETPT</a>	FEAT_MOPS
0	11	0010	<a href="#">SETPN, SETMN, SETEN — SETPN</a>	FEAT_MOPS
0	11	0011	<a href="#">SETPTN, SETMTN, SETETN — SETPTN</a>	FEAT_MOPS
0	11	0100	<a href="#">SETP, SETM, SETE — SETM</a>	FEAT_MOPS

Decode fields			Instruction Details	Feature
o0	op1	op2		
0	11	0101	<a href="#">SETPT, SETMT, SETET — SETMT</a>	FEAT_MOPS
0	11	0110	<a href="#">SETPN, SETMN, SETEN — SETMN</a>	FEAT_MOPS
0	11	0111	<a href="#">SETPTN, SETMTN, SETETN — SETMTN</a>	FEAT_MOPS
0	11	1000	<a href="#">SETP, SETM, SETE — SETE</a>	FEAT_MOPS
0	11	1001	<a href="#">SETPT, SETMT, SETET — SETET</a>	FEAT_MOPS
0	11	1010	<a href="#">SETPN, SETMN, SETEN — SETEN</a>	FEAT_MOPS
0	11	1011	<a href="#">SETPTN, SETMTN, SETETN — SETETN</a>	FEAT_MOPS
0	11	11xx	UNALLOCATED	-
1	00	0000	<a href="#">CPYP, CPYM, CPYE — CPYP</a>	FEAT_MOPS
1	00	0001	<a href="#">CPYPWT, CPYMWI, CPYEWT — CPYPWT</a>	FEAT_MOPS
1	00	0010	<a href="#">CPYPRT, CPYMRT, CPYERT — CPYPRT</a>	FEAT_MOPS
1	00	0011	<a href="#">CPYPT, CPYMT, CPYET — CPYPT</a>	FEAT_MOPS
1	00	0100	<a href="#">CPYPWN, CPYMWN, CPYEWN — CPYPWN</a>	FEAT_MOPS
1	00	0101	<a href="#">CPYPWTWN, CPYMWIWN, CPYEWTWN — CPYPWTWN</a>	FEAT_MOPS
1	00	0110	<a href="#">CPYPRTWN, CPYMRTWN, CPYERTWN — CPYPRTWN</a>	FEAT_MOPS
1	00	0111	<a href="#">CPYPTWN, CPYMTWN, CPYETWN — CPYPTWN</a>	FEAT_MOPS
1	00	1000	<a href="#">CPYPRN, CPYMRN, CPYERN — CPYPRN</a>	FEAT_MOPS
1	00	1001	<a href="#">CPYPWTRN, CPYMWTRN, CPYEWTRN — CPYPWTRN</a>	FEAT_MOPS
1	00	1010	<a href="#">CPYPRTRN, CPYMRTRN, CPYERTRN — CPYPRTRN</a>	FEAT_MOPS
1	00	1011	<a href="#">CPYPTRN, CPYMRN, CPYETRN — CPYPTRN</a>	FEAT_MOPS
1	00	1100	<a href="#">CPYPN, CPYMN, CPYEN — CPYPN</a>	FEAT_MOPS
1	00	1101	<a href="#">CPYPWTN, CPYMWIWN, CPYEWTN — CPYPWTN</a>	FEAT_MOPS
1	00	1110	<a href="#">CPYPRTN, CPYMRN, CPYERTN — CPYPRTN</a>	FEAT_MOPS
1	00	1111	<a href="#">CPYPTN, CPYMTN, CPYETN — CPYPTN</a>	FEAT_MOPS
1	01	0000	<a href="#">CPYP, CPYM, CPYE — CPYM</a>	FEAT_MOPS
1	01	0001	<a href="#">CPYPWT, CPYMWI, CPYEWT — CPYMWI</a>	FEAT_MOPS
1	01	0010	<a href="#">CPYPRT, CPYMRT, CPYERT — CPYMRT</a>	FEAT_MOPS
1	01	0011	<a href="#">CPYPT, CPYMT, CPYET — CPYMT</a>	FEAT_MOPS
1	01	0100	<a href="#">CPYPWN, CPYMWN, CPYEWN — CPYMWN</a>	FEAT_MOPS
1	01	0101	<a href="#">CPYPWTWN, CPYMWIWN, CPYEWTWN — CPYMWIWN</a>	FEAT_MOPS
1	01	0110	<a href="#">CPYPRTWN, CPYMRWN, CPYERTWN — CPYMRWN</a>	FEAT_MOPS
1	01	0111	<a href="#">CPYPTWN, CPYMTWN, CPYETWN — CPYMTWN</a>	FEAT_MOPS
1	01	1000	<a href="#">CPYPRN, CPYMRN, CPYERN — CPYMRN</a>	FEAT_MOPS
1	01	1001	<a href="#">CPYPWTRN, CPYMWTRN, CPYEWTRN — CPYMWTRN</a>	FEAT_MOPS
1	01	1010	<a href="#">CPYPRTRN, CPYMRTRN, CPYERTRN — CPYMRTRN</a>	FEAT_MOPS
1	01	1011	<a href="#">CPYPTRN, CPYMRN, CPYETRN — CPYMRN</a>	FEAT_MOPS
1	01	1100	<a href="#">CPYPN, CPYMN, CPYEN — CPYMN</a>	FEAT_MOPS
1	01	1101	<a href="#">CPYPWTN, CPYMWIWN, CPYEWTN — CPYMWIWN</a>	FEAT_MOPS
1	01	1110	<a href="#">CPYPRTN, CPYMRN, CPYERTN — CPYMRN</a>	FEAT_MOPS
1	01	1111	<a href="#">CPYPTN, CPYMTN, CPYETN — CPYMTN</a>	FEAT_MOPS
1	10	0000	<a href="#">CPYP, CPYM, CPYE — CPYE</a>	FEAT_MOPS
1	10	0001	<a href="#">CPYPWT, CPYMWI, CPYEWT — CPYEWT</a>	FEAT_MOPS
1	10	0010	<a href="#">CPYPRT, CPYMRT, CPYERT — CPYERT</a>	FEAT_MOPS
1	10	0011	<a href="#">CPYPT, CPYMT, CPYET — CPYET</a>	FEAT_MOPS
1	10	0100	<a href="#">CPYPWN, CPYMWN, CPYEWN — CPYEWN</a>	FEAT_MOPS
1	10	0101	<a href="#">CPYPWTWN, CPYMWIWN, CPYEWTWN — CPYEWTWN</a>	FEAT_MOPS
1	10	0110	<a href="#">CPYPRTWN, CPYMRWN, CPYERTWN — CPYERTWN</a>	FEAT_MOPS

Decode fields			Instruction Details	Feature
o0	op1	op2		
1	10	0111	<a href="#">CPYPTWN, CPYMTWN, CPYETWN</a> — <a href="#">CPYETWN</a>	FEAT_MOPS
1	10	1000	<a href="#">CPYPRN, CPYMRN, CPYERN</a> — <a href="#">CPYERN</a>	FEAT_MOPS
1	10	1001	<a href="#">CPYPWTRN, CPYMWTRN, CPYEWTRN</a> — <a href="#">CPYEWTRN</a>	FEAT_MOPS
1	10	1010	<a href="#">CPYPRTN, CPYMRN, CPYERTRN</a> — <a href="#">CPYERTRN</a>	FEAT_MOPS
1	10	1011	<a href="#">CPYPTRN, CPYMTN, CPYETRN</a> — <a href="#">CPYETRN</a>	FEAT_MOPS
1	10	1100	<a href="#">CPYPN, CPYMN, CPYEN</a> — <a href="#">CPYEN</a>	FEAT_MOPS
1	10	1101	<a href="#">CPYPWTN, CPYMWTN, CPYEWTN</a> — <a href="#">CPYEWTN</a>	FEAT_MOPS
1	10	1110	<a href="#">CPYPRTN, CPYMRN, CPYERTN</a> — <a href="#">CPYERTN</a>	FEAT_MOPS
1	10	1111	<a href="#">CPYPTN, CPYMTN, CPYETN</a> — <a href="#">CPYETN</a>	FEAT_MOPS
1	11	0000	<a href="#">SETGP, SETGM, SETGE</a> — <a href="#">SETGP</a>	FEAT_MOPS
1	11	0001	<a href="#">SETGPT, SETGMT, SETGET</a> — <a href="#">SETGPT</a>	FEAT_MOPS
1	11	0010	<a href="#">SETGPN, SETGMN, SETGEN</a> — <a href="#">SETGPN</a>	FEAT_MOPS
1	11	0011	<a href="#">SETGPTN, SETGMTN, SETGETN</a> — <a href="#">SETGPTN</a>	FEAT_MOPS
1	11	0100	<a href="#">SETGP, SETGM, SETGE</a> — <a href="#">SETGM</a>	FEAT_MOPS
1	11	0101	<a href="#">SETGPT, SETGMT, SETGET</a> — <a href="#">SETGMT</a>	FEAT_MOPS
1	11	0110	<a href="#">SETGPN, SETGMN, SETGEN</a> — <a href="#">SETGMN</a>	FEAT_MOPS
1	11	0111	<a href="#">SETGPTN, SETGMTN, SETGETN</a> — <a href="#">SETGMTN</a>	FEAT_MOPS
1	11	1000	<a href="#">SETGP, SETGM, SETGE</a> — <a href="#">SETGE</a>	FEAT_MOPS
1	11	1001	<a href="#">SETGPT, SETGMT, SETGET</a> — <a href="#">SETGET</a>	FEAT_MOPS
1	11	1010	<a href="#">SETGPN, SETGMN, SETGEN</a> — <a href="#">SETGEN</a>	FEAT_MOPS
1	11	1011	<a href="#">SETGPTN, SETGMTN, SETGETN</a> — <a href="#">SETGETN</a>	FEAT_MOPS
1	11	11xx	UNALLOCATED	-

**Load/store no-allocate pair (offset)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	VR	0	0	0	L	imm7							Rt2					Rn				Rt					

Decode fields			Instruction Details
opc	VR	L	
00	0	0	<a href="#">STNP</a> — 32-bit
00	0	1	<a href="#">LDNP</a> — 32-bit
00	1	0	<a href="#">STNP (SIMD&amp;FP)</a> — 32-bit
00	1	1	<a href="#">LDNP (SIMD&amp;FP)</a> — 32-bit
01	0		UNALLOCATED
01	1	0	<a href="#">STNP (SIMD&amp;FP)</a> — 64-bit
01	1	1	<a href="#">LDNP (SIMD&amp;FP)</a> — 64-bit
10	0	0	<a href="#">STNP</a> — 64-bit
10	0	1	<a href="#">LDNP</a> — 64-bit
10	1	0	<a href="#">STNP (SIMD&amp;FP)</a> — 128-bit
10	1	1	<a href="#">LDNP (SIMD&amp;FP)</a> — 128-bit
11			UNALLOCATED

**Load/store register pair (post-indexed)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc	1	0	1	VR	0	0	1	L	imm7							Rt2				Rn				Rt							

Decode fields			Instruction Details		Feature
opc	VR	L			
00	0	0	<a href="#">STP — 32-bit</a>		-
00	0	1	<a href="#">LDP — 32-bit</a>		-
00	1	0	<a href="#">STP (SIMD&amp;FP) — 32-bit</a>		-
00	1	1	<a href="#">LDP (SIMD&amp;FP) — 32-bit</a>		-
01	0	0	<a href="#">STGP</a>		FEAT_MTE
01	0	1	<a href="#">LDPSW</a>		-
01	1	0	<a href="#">STP (SIMD&amp;FP) — 64-bit</a>		-
01	1	1	<a href="#">LDP (SIMD&amp;FP) — 64-bit</a>		-
10	0	0	<a href="#">STP — 64-bit</a>		-
10	0	1	<a href="#">LDP — 64-bit</a>		-
10	1	0	<a href="#">STP (SIMD&amp;FP) — 128-bit</a>		-
10	1	1	<a href="#">LDP (SIMD&amp;FP) — 128-bit</a>		-
11			UNALLOCATED		-

**Load/store register pair (offset)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc	1	0	1	VR	0	1	0	L	imm7							Rt2				Rn				Rt							

Decode fields			Instruction Details		Feature
opc	VR	L			
00	0	0	<a href="#">STP — 32-bit</a>		-
00	0	1	<a href="#">LDP — 32-bit</a>		-
00	1	0	<a href="#">STP (SIMD&amp;FP) — 32-bit</a>		-
00	1	1	<a href="#">LDP (SIMD&amp;FP) — 32-bit</a>		-
01	0	0	<a href="#">STGP</a>		FEAT_MTE
01	0	1	<a href="#">LDPSW</a>		-
01	1	0	<a href="#">STP (SIMD&amp;FP) — 64-bit</a>		-
01	1	1	<a href="#">LDP (SIMD&amp;FP) — 64-bit</a>		-
10	0	0	<a href="#">STP — 64-bit</a>		-
10	0	1	<a href="#">LDP — 64-bit</a>		-
10	1	0	<a href="#">STP (SIMD&amp;FP) — 128-bit</a>		-
10	1	1	<a href="#">LDP (SIMD&amp;FP) — 128-bit</a>		-
11			UNALLOCATED		-

**Load/store register pair (pre-indexed)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc	1	0	1	VR	0	1	1	L	imm7							Rt2				Rn				Rt							

Decode fields			Instruction Details		Feature
opc	VR	L			
00	0	0	<a href="#">STP — 32-bit</a>		-
00	0	1	<a href="#">LDP — 32-bit</a>		-
00	1	0	<a href="#">STP (SIMD&amp;FP) — 32-bit</a>		-

Decode fields			Instruction Details	Feature
opc	VR	L		
00	1	1	<a href="#">LDP (SIMD&amp;FP) — 32-bit</a>	-
01	0	0	<a href="#">STGP</a>	FEAT_MTE
01	0	1	<a href="#">LDPSW</a>	-
01	1	0	<a href="#">STP (SIMD&amp;FP) — 64-bit</a>	-
01	1	1	<a href="#">LDP (SIMD&amp;FP) — 64-bit</a>	-
10	0	0	<a href="#">STP — 64-bit</a>	-
10	0	1	<a href="#">LDP — 64-bit</a>	-
10	1	0	<a href="#">STP (SIMD&amp;FP) — 128-bit</a>	-
10	1	1	<a href="#">LDP (SIMD&amp;FP) — 128-bit</a>	-
11			UNALLOCATED	-

**Load/store register (unscaled immediate)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	VR	0	0	opc		0	imm9										0	0	Rn				Rt				

Decode fields			Instruction Details
size	VR	opc	
x1	1	1x	UNALLOCATED
00	0	00	<a href="#">STURB</a>
00	0	01	<a href="#">LDURB</a>
00	0	10	<a href="#">LDURSB — 64-bit</a>
00	0	11	<a href="#">LDURSB — 32-bit</a>
00	1	00	<a href="#">STUR (SIMD&amp;FP) — 8-bit</a>
00	1	01	<a href="#">LDUR (SIMD&amp;FP) — 8-bit</a>
00	1	10	<a href="#">STUR (SIMD&amp;FP) — 128-bit</a>
00	1	11	<a href="#">LDUR (SIMD&amp;FP) — 128-bit</a>
01	0	00	<a href="#">STURH</a>
01	0	01	<a href="#">LDURH</a>
01	0	10	<a href="#">LDURSH — 64-bit</a>
01	0	11	<a href="#">LDURSH — 32-bit</a>
01	1	00	<a href="#">STUR (SIMD&amp;FP) — 16-bit</a>
01	1	01	<a href="#">LDUR (SIMD&amp;FP) — 16-bit</a>
1x	0	11	UNALLOCATED
1x	1	1x	UNALLOCATED
10	0	00	<a href="#">STUR — 32-bit</a>
10	0	01	<a href="#">LDUR — 32-bit</a>
10	0	10	<a href="#">LDURSW</a>
10	1	00	<a href="#">STUR (SIMD&amp;FP) — 32-bit</a>
10	1	01	<a href="#">LDUR (SIMD&amp;FP) — 32-bit</a>
11	0	00	<a href="#">STUR — 64-bit</a>
11	0	01	<a href="#">LDUR — 64-bit</a>
11	0	10	<a href="#">PRFUM</a>
11	1	00	<a href="#">STUR (SIMD&amp;FP) — 64-bit</a>
11	1	01	<a href="#">LDUR (SIMD&amp;FP) — 64-bit</a>



**Load/store register (immediate post-indexed)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	VR	0	0	opc		0	imm9									0	1	Rn				Rt					

Decode fields			Instruction Details
size	VR	opc	
×1	1	1×	UNALLOCATED
00	0	00	<a href="#">STRB (immediate)</a>
00	0	01	<a href="#">LDRB (immediate)</a>
00	0	10	<a href="#">LDRSB (immediate)</a> — 64-bit
00	0	11	<a href="#">LDRSB (immediate)</a> — 32-bit
00	1	00	<a href="#">STR (immediate, SIMD&amp;FP)</a> — 8-bit
00	1	01	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — 8-bit
00	1	10	<a href="#">STR (immediate, SIMD&amp;FP)</a> — 128-bit
00	1	11	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — 128-bit
01	0	00	<a href="#">STRH (immediate)</a>
01	0	01	<a href="#">LDRH (immediate)</a>
01	0	10	<a href="#">LDRSH (immediate)</a> — 64-bit
01	0	11	<a href="#">LDRSH (immediate)</a> — 32-bit
01	1	00	<a href="#">STR (immediate, SIMD&amp;FP)</a> — 16-bit
01	1	01	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — 16-bit
1×	0	11	UNALLOCATED
1×	1	1×	UNALLOCATED
10	0	00	<a href="#">STR (immediate)</a> — 32-bit
10	0	01	<a href="#">LDR (immediate)</a> — 32-bit
10	0	10	<a href="#">LDRSW (immediate)</a>
10	1	00	<a href="#">STR (immediate, SIMD&amp;FP)</a> — 32-bit
10	1	01	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — 32-bit
11	0	00	<a href="#">STR (immediate)</a> — 64-bit
11	0	01	<a href="#">LDR (immediate)</a> — 64-bit
11	0	10	UNALLOCATED
11	1	00	<a href="#">STR (immediate, SIMD&amp;FP)</a> — 64-bit
11	1	01	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — 64-bit

**Load/store register (unprivileged)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	VR	0	0	opc		0	imm9									1	0	Rn				Rt					

Decode fields			Instruction Details
size	VR	opc	
	1		UNALLOCATED
00	0	00	<a href="#">STTRB</a>
00	0	01	<a href="#">LDTRB</a>
00	0	10	<a href="#">LDTRSB</a> — 64-bit
00	0	11	<a href="#">LDTRSB</a> — 32-bit
01	0	00	<a href="#">STTRH</a>
01	0	01	<a href="#">LDTRH</a>

Decode fields			Instruction Details
size	VR	opc	
01	0	10	<a href="#">LDTRSH — 64-bit</a>
01	0	11	<a href="#">LDTRSH — 32-bit</a>
1x	0	11	UNALLOCATED
10	0	00	<a href="#">STTR — 32-bit</a>
10	0	01	<a href="#">LDTR — 32-bit</a>
10	0	10	<a href="#">LDTRSW</a>
11	0	00	<a href="#">STTR — 64-bit</a>
11	0	01	<a href="#">LDTR — 64-bit</a>
11	0	10	UNALLOCATED

**Load/store register (immediate pre-indexed)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	VR	0	0	opc		0	imm9										1	1	Rn				Rt				

Decode fields			Instruction Details
size	VR	opc	
x1	1	1x	UNALLOCATED
00	0	00	<a href="#">STRB (immediate)</a>
00	0	01	<a href="#">LDRB (immediate)</a>
00	0	10	<a href="#">LDRSB (immediate) — 64-bit</a>
00	0	11	<a href="#">LDRSB (immediate) — 32-bit</a>
00	1	00	<a href="#">STR (immediate, SIMD&amp;FP) — 8-bit</a>
00	1	01	<a href="#">LDR (immediate, SIMD&amp;FP) — 8-bit</a>
00	1	10	<a href="#">STR (immediate, SIMD&amp;FP) — 128-bit</a>
00	1	11	<a href="#">LDR (immediate, SIMD&amp;FP) — 128-bit</a>
01	0	00	<a href="#">STRH (immediate)</a>
01	0	01	<a href="#">LDRH (immediate)</a>
01	0	10	<a href="#">LDRSH (immediate) — 64-bit</a>
01	0	11	<a href="#">LDRSH (immediate) — 32-bit</a>
01	1	00	<a href="#">STR (immediate, SIMD&amp;FP) — 16-bit</a>
01	1	01	<a href="#">LDR (immediate, SIMD&amp;FP) — 16-bit</a>
1x	0	11	UNALLOCATED
1x	1	1x	UNALLOCATED
10	0	00	<a href="#">STR (immediate) — 32-bit</a>
10	0	01	<a href="#">LDR (immediate) — 32-bit</a>
10	0	10	<a href="#">LDRSW (immediate)</a>
10	1	00	<a href="#">STR (immediate, SIMD&amp;FP) — 32-bit</a>
10	1	01	<a href="#">LDR (immediate, SIMD&amp;FP) — 32-bit</a>
11	0	00	<a href="#">STR (immediate) — 64-bit</a>
11	0	01	<a href="#">LDR (immediate) — 64-bit</a>
11	0	10	UNALLOCATED
11	1	00	<a href="#">STR (immediate, SIMD&amp;FP) — 64-bit</a>
11	1	01	<a href="#">LDR (immediate, SIMD&amp;FP) — 64-bit</a>

## Atomic memory operations

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	VR	0	0	A	R	1	Rs			o3		opc		0		0	Rn			Rt							

size	VR	Decode fields			Rs	o3	opc	Instruction Details	Feature
	0					1	11x	UNALLOCATED	-
	0	0				1	100	UNALLOCATED	-
	0	0	1			1	101	UNALLOCATED	-
	0	1	0			1	101	UNALLOCATED	-
	0	1	1			1	100	UNALLOCATED	-
	0	1	1			1	101	UNALLOCATED	-
	1							UNALLOCATED	-
00	0	0	0			0	000	<a href="#">LDADDB</a> , <a href="#">LDADDAB</a> , <a href="#">LDADDALB</a> , <a href="#">LDADDLB</a> — <a href="#">LDADDB</a>	FEAT_LSE
00	0	0	0			0	001	<a href="#">LDCLRB</a> , <a href="#">LDCLRAB</a> , <a href="#">LDCLRALB</a> , <a href="#">LDCLRLB</a> — <a href="#">LDCLRB</a>	FEAT_LSE
00	0	0	0			0	010	<a href="#">LDEORB</a> , <a href="#">LDEORAB</a> , <a href="#">LDEORALB</a> , <a href="#">LDEORLB</a> — <a href="#">LDEORB</a>	FEAT_LSE
00	0	0	0			0	011	<a href="#">LDSETB</a> , <a href="#">LDSETAB</a> , <a href="#">LDSETALB</a> , <a href="#">LDSETLB</a> — <a href="#">LDSETB</a>	FEAT_LSE
00	0	0	0			0	100	<a href="#">LDSMAXB</a> , <a href="#">LDSMAXAB</a> , <a href="#">LDSMAXALB</a> , <a href="#">LDSMAXLB</a> — <a href="#">LDSMAXB</a>	FEAT_LSE
00	0	0	0			0	101	<a href="#">LDSMINB</a> , <a href="#">LDSMINAB</a> , <a href="#">LDSMINALB</a> , <a href="#">LDSMINLB</a> — <a href="#">LDSMINB</a>	FEAT_LSE
00	0	0	0			0	110	<a href="#">LDUMAXB</a> , <a href="#">LDUMAXAB</a> , <a href="#">LDUMAXALB</a> , <a href="#">LDUMAXLB</a> — <a href="#">LDUMAXB</a>	FEAT_LSE
00	0	0	0			0	111	<a href="#">LDUMINB</a> , <a href="#">LDUMINAB</a> , <a href="#">LDUMINALB</a> , <a href="#">LDUMINLB</a> — <a href="#">LDUMINB</a>	FEAT_LSE
00	0	0	0			1	000	<a href="#">SWPB</a> , <a href="#">SWPAB</a> , <a href="#">SWPALB</a> , <a href="#">SWPLB</a> — <a href="#">SWPB</a>	FEAT_LSE
00	0	0	0			1	001	<a href="#">RCWCLR</a> , <a href="#">RCWCLRA</a> , <a href="#">RCWCLRL</a> , <a href="#">RCWCLRAL</a> — <a href="#">RCWCLR</a>	FEAT_THE
00	0	0	0			1	010	<a href="#">RCWSWP</a> , <a href="#">RCWSWPA</a> , <a href="#">RCWSWPL</a> , <a href="#">RCWSWPAL</a> — <a href="#">RCWSWP</a>	FEAT_THE
00	0	0	0			1	011	<a href="#">RCWSET</a> , <a href="#">RCWSETA</a> , <a href="#">RCWSETL</a> , <a href="#">RCWSETAL</a> — <a href="#">RCWSET</a>	FEAT_THE
00	0	0	0			1	101	UNALLOCATED	-
00	0	0	1			0	000	<a href="#">LDADDB</a> , <a href="#">LDADDAB</a> , <a href="#">LDADDALB</a> , <a href="#">LDADDLB</a> — <a href="#">LDADDLB</a>	FEAT_LSE
00	0	0	1			0	001	<a href="#">LDCLRB</a> , <a href="#">LDCLRAB</a> , <a href="#">LDCLRALB</a> , <a href="#">LDCLRLB</a> — <a href="#">LDCLRLB</a>	FEAT_LSE
00	0	0	1			0	010	<a href="#">LDEORB</a> , <a href="#">LDEORAB</a> , <a href="#">LDEORALB</a> , <a href="#">LDEORLB</a> — <a href="#">LDEORLB</a>	FEAT_LSE
00	0	0	1			0	011	<a href="#">LDSETB</a> , <a href="#">LDSETAB</a> , <a href="#">LDSETALB</a> , <a href="#">LDSETLB</a> — <a href="#">LDSETLB</a>	FEAT_LSE
00	0	0	1			0	100	<a href="#">LDSMAXB</a> , <a href="#">LDSMAXAB</a> , <a href="#">LDSMAXALB</a> , <a href="#">LDSMAXLB</a> — <a href="#">LDSMAXLB</a>	FEAT_LSE
00	0	0	1			0	101	<a href="#">LDSMINB</a> , <a href="#">LDSMINAB</a> , <a href="#">LDSMINALB</a> , <a href="#">LDSMINLB</a> — <a href="#">LDSMINLB</a>	FEAT_LSE
00	0	0	1			0	110	<a href="#">LDUMAXB</a> , <a href="#">LDUMAXAB</a> , <a href="#">LDUMAXALB</a> , <a href="#">LDUMAXLB</a> — <a href="#">LDUMAXLB</a>	FEAT_LSE
00	0	0	1			0	111	<a href="#">LDUMINB</a> , <a href="#">LDUMINAB</a> , <a href="#">LDUMINALB</a> , <a href="#">LDUMINLB</a> — <a href="#">LDUMINLB</a>	FEAT_LSE

size	VR	Decode fields			o3	opc	Instruction Details	Feature
		A	R	Rs				
00	0	0	1		1	000	<a href="#">SWPB, SWPAB, SWPALB, SWPLB — SWPLB</a>	FEAT_LSE
00	0	0	1		1	001	<a href="#">RCWCLR, RCWCLRA, RCWCLRL, RCWCLRAL — RCWCLRL</a>	FEAT_THE
00	0	0	1		1	010	<a href="#">RCWSWP, RCWSWPA, RCWSWPL, RCWSWPAL — RCWSWPL</a>	FEAT_THE
00	0	0	1		1	011	<a href="#">RCWSET, RCWSETA, RCWSETL, RCWSETAL — RCWSETL</a>	FEAT_THE
00	0	1	0		0	000	<a href="#">LDADDB, LDADDAB, LDADDALB, LDADDLB — LDADDAB</a>	FEAT_LSE
00	0	1	0		0	001	<a href="#">LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — LDCLRAB</a>	FEAT_LSE
00	0	1	0		0	010	<a href="#">LDEORB, LDEORAB, LDEORALB, LDEORLB — LDEORAB</a>	FEAT_LSE
00	0	1	0		0	011	<a href="#">LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETAB</a>	FEAT_LSE
00	0	1	0		0	100	<a href="#">LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXAB</a>	FEAT_LSE
00	0	1	0		0	101	<a href="#">LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINAB</a>	FEAT_LSE
00	0	1	0		0	110	<a href="#">LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXAB</a>	FEAT_LSE
00	0	1	0		0	111	<a href="#">LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINAB</a>	FEAT_LSE
00	0	1	0		1	000	<a href="#">SWPB, SWPAB, SWPALB, SWPLB — SWPAB</a>	FEAT_LSE
00	0	1	0		1	001	<a href="#">RCWCLR, RCWCLRA, RCWCLRL, RCWCLRAL — RCWCLRA</a>	FEAT_THE
00	0	1	0		1	010	<a href="#">RCWSWP, RCWSWPA, RCWSWPL, RCWSWPAL — RCWSWPA</a>	FEAT_THE
00	0	1	0		1	011	<a href="#">RCWSET, RCWSETA, RCWSETL, RCWSETAL — RCWSETA</a>	FEAT_THE
00	0	1	0		1	100	<a href="#">LDAPRB</a>	FEAT_LRCPC
00	0	1	1		0	000	<a href="#">LDADDB, LDADDAB, LDADDALB, LDADDLB — LDADDALB</a>	FEAT_LSE
00	0	1	1		0	001	<a href="#">LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — LDCLRALB</a>	FEAT_LSE
00	0	1	1		0	010	<a href="#">LDEORB, LDEORAB, LDEORALB, LDEORLB — LDEORALB</a>	FEAT_LSE
00	0	1	1		0	011	<a href="#">LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETALB</a>	FEAT_LSE
00	0	1	1		0	100	<a href="#">LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXALB</a>	FEAT_LSE
00	0	1	1		0	101	<a href="#">LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINALB</a>	FEAT_LSE
00	0	1	1		0	110	<a href="#">LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXALB</a>	FEAT_LSE
00	0	1	1		0	111	<a href="#">LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINALB</a>	FEAT_LSE
00	0	1	1		1	000	<a href="#">SWPB, SWPAB, SWPALB, SWPLB — SWPALB</a>	FEAT_LSE
00	0	1	1		1	001	<a href="#">RCWCLR, RCWCLRA, RCWCLRL, RCWCLRAL — RCWCLRAL</a>	FEAT_THE
00	0	1	1		1	010	<a href="#">RCWSWP, RCWSWPA, RCWSWPL, RCWSWPAL — RCWSWPAL</a>	FEAT_THE
00	0	1	1		1	011	<a href="#">RCWSET, RCWSETA, RCWSETL, RCWSETAL — RCWSETAL</a>	FEAT_THE

size	VR	Decode fields			o3	opc	Instruction Details	Feature
		A	R	Rs				
01	0	0	0		0	000	<a href="#">LDADDH, LDADDAH, LDADDALH, LDADDLH</a> — <a href="#">LDADDH</a>	FEAT_LSE
01	0	0	0		0	001	<a href="#">LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH</a> — <a href="#">LDCLRH</a>	FEAT_LSE
01	0	0	0		0	010	<a href="#">LDEORH, LDEORAH, LDEORALH, LDEORLH</a> — <a href="#">LDEORH</a>	FEAT_LSE
01	0	0	0		0	011	<a href="#">LDSETH, LDSETAH, LDSETALH, LDSETLH</a> — <a href="#">LDSETH</a>	FEAT_LSE
01	0	0	0		0	100	<a href="#">LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH</a> — <a href="#">LDSMAXH</a>	FEAT_LSE
01	0	0	0		0	101	<a href="#">LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH</a> — <a href="#">LDSMINH</a>	FEAT_LSE
01	0	0	0		0	110	<a href="#">LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH</a> — <a href="#">LDUMAXH</a>	FEAT_LSE
01	0	0	0		0	111	<a href="#">LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH</a> — <a href="#">LDUMINH</a>	FEAT_LSE
01	0	0	0		1	000	<a href="#">SWPH, SWPAH, SWPALH, SWPLH</a> — <a href="#">SWPH</a>	FEAT_LSE
01	0	0	0		1	001	<a href="#">RCWSCLR, RCWSCLRA, RCWSCLRL, RCWSCLRAL</a> — <a href="#">RCWSCLR</a>	FEAT_THE
01	0	0	0		1	010	<a href="#">RCWSSWP, RCWSSWPA, RCWSSWPL, RCWSSWPAL</a> — <a href="#">RCWSSWP</a>	FEAT_THE
01	0	0	0		1	011	<a href="#">RCWSSET, RCWSSETA, RCWSSETL, RCWSSETAL</a> — <a href="#">RCWSSET</a>	FEAT_THE
01	0	0	0		1	101	UNALLOCATED	-
01	0	0	1		0	000	<a href="#">LDADDH, LDADDAH, LDADDALH, LDADDLH</a> — <a href="#">LDADDLH</a>	FEAT_LSE
01	0	0	1		0	001	<a href="#">LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH</a> — <a href="#">LDCLRLH</a>	FEAT_LSE
01	0	0	1		0	010	<a href="#">LDEORH, LDEORAH, LDEORALH, LDEORLH</a> — <a href="#">LDEORLH</a>	FEAT_LSE
01	0	0	1		0	011	<a href="#">LDSETH, LDSETAH, LDSETALH, LDSETLH</a> — <a href="#">LDSETLH</a>	FEAT_LSE
01	0	0	1		0	100	<a href="#">LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH</a> — <a href="#">LDSMAXLH</a>	FEAT_LSE
01	0	0	1		0	101	<a href="#">LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH</a> — <a href="#">LDSMINLH</a>	FEAT_LSE
01	0	0	1		0	110	<a href="#">LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH</a> — <a href="#">LDUMAXLH</a>	FEAT_LSE
01	0	0	1		0	111	<a href="#">LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH</a> — <a href="#">LDUMINLH</a>	FEAT_LSE
01	0	0	1		1	000	<a href="#">SWPH, SWPAH, SWPALH, SWPLH</a> — <a href="#">SWPLH</a>	FEAT_LSE
01	0	0	1		1	001	<a href="#">RCWSCLR, RCWSCLRA, RCWSCLRL, RCWSCLRAL</a> — <a href="#">RCWSCLRL</a>	FEAT_THE
01	0	0	1		1	010	<a href="#">RCWSSWP, RCWSSWPA, RCWSSWPL, RCWSSWPAL</a> — <a href="#">RCWSSWPL</a>	FEAT_THE
01	0	0	1		1	011	<a href="#">RCWSSET, RCWSSETA, RCWSSETL, RCWSSETAL</a> — <a href="#">RCWSSETL</a>	FEAT_THE
01	0	1	0		0	000	<a href="#">LDADDH, LDADDAH, LDADDALH, LDADDLH</a> — <a href="#">LDADDAH</a>	FEAT_LSE
01	0	1	0		0	001	<a href="#">LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH</a> — <a href="#">LDCLRAH</a>	FEAT_LSE

size	VR	Decode fields			o3	opc	Instruction Details	Feature
		A	R	Rs				
01	0	1	0		0	010	<a href="#">LDEORH, LDEORAH, LDEORALH, LDEORLH</a> — <a href="#">LDEORAH</a>	FEAT_LSE
01	0	1	0		0	011	<a href="#">LDSETH, LDSETAH, LDSETALH, LDSETLH</a> — <a href="#">LDSETAH</a>	FEAT_LSE
01	0	1	0		0	100	<a href="#">LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH</a> — <a href="#">LDSMAXAH</a>	FEAT_LSE
01	0	1	0		0	101	<a href="#">LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH</a> — <a href="#">LDSMINAH</a>	FEAT_LSE
01	0	1	0		0	110	<a href="#">LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH</a> — <a href="#">LDUMAXAH</a>	FEAT_LSE
01	0	1	0		0	111	<a href="#">LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH</a> — <a href="#">LDUMINAH</a>	FEAT_LSE
01	0	1	0		1	000	<a href="#">SWPH, SWPAH, SWPALH, SWPLH</a> — <a href="#">SWPAH</a>	FEAT_LSE
01	0	1	0		1	001	<a href="#">RCWSCLR, RCWSCLRA, RCWSCLRL, RCWSCLRAL</a> — <a href="#">RCWSCLRA</a>	FEAT_THE
01	0	1	0		1	010	<a href="#">RCWSSWP, RCWSSWPA, RCWSSWPL, RCWSSWPAL</a> — <a href="#">RCWSSWPA</a>	FEAT_THE
01	0	1	0		1	011	<a href="#">RCWSSET, RCWSSETA, RCWSSETL, RCWSSETAL</a> — <a href="#">RCWSSETA</a>	FEAT_THE
01	0	1	0		1	100	<a href="#">LDAPRH</a>	FEAT_LRCPC
01	0	1	1		0	000	<a href="#">LDADDH, LDADDAH, LDADDALH, LDADDLH</a> — <a href="#">LDADDALH</a>	FEAT_LSE
01	0	1	1		0	001	<a href="#">LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH</a> — <a href="#">LDCLRALH</a>	FEAT_LSE
01	0	1	1		0	010	<a href="#">LDEORH, LDEORAH, LDEORALH, LDEORLH</a> — <a href="#">LDEORALH</a>	FEAT_LSE
01	0	1	1		0	011	<a href="#">LDSETH, LDSETAH, LDSETALH, LDSETLH</a> — <a href="#">LDSETALH</a>	FEAT_LSE
01	0	1	1		0	100	<a href="#">LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH</a> — <a href="#">LDSMAXALH</a>	FEAT_LSE
01	0	1	1		0	101	<a href="#">LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH</a> — <a href="#">LDSMINALH</a>	FEAT_LSE
01	0	1	1		0	110	<a href="#">LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH</a> — <a href="#">LDUMAXALH</a>	FEAT_LSE
01	0	1	1		0	111	<a href="#">LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH</a> — <a href="#">LDUMINALH</a>	FEAT_LSE
01	0	1	1		1	000	<a href="#">SWPH, SWPAH, SWPALH, SWPLH</a> — <a href="#">SWPALH</a>	FEAT_LSE
01	0	1	1		1	001	<a href="#">RCWSCLR, RCWSCLRA, RCWSCLRL, RCWSCLRAL</a> — <a href="#">RCWSCLRAL</a>	FEAT_THE
01	0	1	1		1	010	<a href="#">RCWSSWP, RCWSSWPA, RCWSSWPL, RCWSSWPAL</a> — <a href="#">RCWSSWPAL</a>	FEAT_THE
01	0	1	1		1	011	<a href="#">RCWSSET, RCWSSETA, RCWSSETL, RCWSSETAL</a> — <a href="#">RCWSSETAL</a>	FEAT_THE
1x	0	0	1		1	001	UNALLOCATED	-
1x	0	0	1		1	010	UNALLOCATED	-
1x	0	0	1		1	011	UNALLOCATED	-
1x	0	1	0		1	001	UNALLOCATED	-
1x	0	1	0		1	010	UNALLOCATED	-
1x	0	1	0		1	011	UNALLOCATED	-
1x	0	1	1		1	001	UNALLOCATED	-

size	VR	Decode fields			o3	opc	Instruction Details	Feature
		A	R	Rs				
1x	0	1	1		1	010	UNALLOCATED	-
1x	0	1	1		1	011	UNALLOCATED	-
10	0	0	0		0	000	<a href="#">LDADD, LDADDA, LDADDAL, LDADDL</a> — 32-bit LDADD	FEAT_LSE
10	0	0	0		0	001	<a href="#">LDCLR, LDCLRA, LDCLRAL, LDCLRL</a> — 32-bit LDCLR	FEAT_LSE
10	0	0	0		0	010	<a href="#">LDEOR, LDEORA, LDEORAL, LDEORL</a> — 32-bit LDEOR	FEAT_LSE
10	0	0	0		0	011	<a href="#">LDSET, LDSETA, LDSETAL, LDSETL</a> — 32-bit LDSET	FEAT_LSE
10	0	0	0		0	100	<a href="#">LDSMAX, LDSMAXA, LDSMAXAL,</a> <a href="#">LDSMAXL</a> — 32-bit LDSMAX	FEAT_LSE
10	0	0	0		0	101	<a href="#">LDSMIN, LDSMINA, LDSMINAL,</a> <a href="#">LDSMINL</a> — 32-bit LDSMIN	FEAT_LSE
10	0	0	0		0	110	<a href="#">LDUMAX, LDUMAXA, LDUMAXAL,</a> <a href="#">LDUMAXL</a> — 32-bit LDUMAX	FEAT_LSE
10	0	0	0		0	111	<a href="#">LDUMIN, LDUMINA, LDUMINAL,</a> <a href="#">LDUMINL</a> — 32-bit LDUMIN	FEAT_LSE
10	0	0	0		1	000	<a href="#">SWP, SWPA, SWPAL, SWPL</a> — 32-bit SWP	FEAT_LSE
10	0	0	0		1	001	UNALLOCATED	-
10	0	0	0		1	010	UNALLOCATED	-
10	0	0	0		1	011	UNALLOCATED	-
10	0	0	0		1	101	UNALLOCATED	-
10	0	0	1		0	000	<a href="#">LDADD, LDADDA, LDADDAL, LDADDL</a> — 32-bit LDADDL	FEAT_LSE
10	0	0	1		0	001	<a href="#">LDCLR, LDCLRA, LDCLRAL, LDCLRL</a> — 32-bit LDCLRL	FEAT_LSE
10	0	0	1		0	010	<a href="#">LDEOR, LDEORA, LDEORAL, LDEORL</a> — 32-bit LDEORL	FEAT_LSE
10	0	0	1		0	011	<a href="#">LDSET, LDSETA, LDSETAL, LDSETL</a> — 32-bit LDSETL	FEAT_LSE
10	0	0	1		0	100	<a href="#">LDSMAX, LDSMAXA, LDSMAXAL,</a> <a href="#">LDSMAXL</a> — 32-bit LDSMAXL	FEAT_LSE
10	0	0	1		0	101	<a href="#">LDSMIN, LDSMINA, LDSMINAL,</a> <a href="#">LDSMINL</a> — 32-bit LDSMINL	FEAT_LSE
10	0	0	1		0	110	<a href="#">LDUMAX, LDUMAXA, LDUMAXAL,</a> <a href="#">LDUMAXL</a> — 32-bit LDUMAXL	FEAT_LSE
10	0	0	1		0	111	<a href="#">LDUMIN, LDUMINA, LDUMINAL,</a> <a href="#">LDUMINL</a> — 32-bit LDUMINL	FEAT_LSE
10	0	0	1		1	000	<a href="#">SWP, SWPA, SWPAL, SWPL</a> — 32-bit SWPL	FEAT_LSE
10	0	1	0		0	000	<a href="#">LDADD, LDADDA, LDADDAL, LDADDL</a> — 32-bit LDADDA	FEAT_LSE
10	0	1	0		0	001	<a href="#">LDCLR, LDCLRA, LDCLRAL, LDCLRL</a> — 32-bit LDCLRA	FEAT_LSE
10	0	1	0		0	010	<a href="#">LDEOR, LDEORA, LDEORAL, LDEORL</a> — 32-bit LDEORA	FEAT_LSE
10	0	1	0		0	011	<a href="#">LDSET, LDSETA, LDSETAL, LDSETL</a> — 32-bit LDSETA	FEAT_LSE
10	0	1	0		0	100	<a href="#">LDSMAX, LDSMAXA, LDSMAXAL,</a> <a href="#">LDSMAXL</a> — 32-bit LDSMAXA	FEAT_LSE
10	0	1	0		0	101	<a href="#">LDSMIN, LDSMINA, LDSMINAL,</a> <a href="#">LDSMINL</a> — 32-bit LDSMINA	FEAT_LSE
10	0	1	0		0	110	<a href="#">LDUMAX, LDUMAXA, LDUMAXAL,</a> <a href="#">LDUMAXL</a> — 32-bit LDUMAXA	FEAT_LSE
10	0	1	0		0	111	<a href="#">LDUMIN, LDUMINA, LDUMINAL,</a> <a href="#">LDUMINL</a> — 32-bit LDUMINA	FEAT_LSE

size	VR	Decode fields			o3	opc	Instruction Details	Feature
		A	R	Rs				
10	0	1	0		1	000	<a href="#">SWP, SWPA, SWPAL, SWPL</a> — 32-bit SWPA	FEAT_LSE
10	0	1	0		1	100	<a href="#">LDAPR</a> — 32-bit	FEAT_LRCPC
10	0	1	1		0	000	<a href="#">LDADD, LDADDA, LDADDAL, LDADDL</a> — 32-bit LDADDAL	FEAT_LSE
10	0	1	1		0	001	<a href="#">LDCLR, LDCLRA, LDCLRAL, LDCLRL</a> — 32-bit LDCLRAL	FEAT_LSE
10	0	1	1		0	010	<a href="#">LDEOR, LDEORA, LDEORAL, LDEORL</a> — 32-bit LDEORAL	FEAT_LSE
10	0	1	1		0	011	<a href="#">LDSET, LDSETA, LDSETAL, LDSETL</a> — 32-bit LDSETAL	FEAT_LSE
10	0	1	1		0	100	<a href="#">LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL</a> — 32-bit LDSMAXAL	FEAT_LSE
10	0	1	1		0	101	<a href="#">LDSMIN, LDSMINA, LDSMINAL, LDSMINL</a> — 32-bit LDSMINAL	FEAT_LSE
10	0	1	1		0	110	<a href="#">LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL</a> — 32-bit LDUMAXAL	FEAT_LSE
10	0	1	1		0	111	<a href="#">LDUMIN, LDUMINA, LDUMINAL, LDUMINL</a> — 32-bit LDUMINAL	FEAT_LSE
10	0	1	1		1	000	<a href="#">SWP, SWPA, SWPAL, SWPL</a> — 32-bit SWPAL	FEAT_LSE
11	0	0	0		0	000	<a href="#">LDADD, LDADDA, LDADDAL, LDADDL</a> — 64-bit LDADD	FEAT_LSE
11	0	0	0		0	001	<a href="#">LDCLR, LDCLRA, LDCLRAL, LDCLRL</a> — 64-bit LDCLR	FEAT_LSE
11	0	0	0		0	010	<a href="#">LDEOR, LDEORA, LDEORAL, LDEORL</a> — 64-bit LDEOR	FEAT_LSE
11	0	0	0		0	011	<a href="#">LDSET, LDSETA, LDSETAL, LDSETL</a> — 64-bit LDSET	FEAT_LSE
11	0	0	0		0	100	<a href="#">LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL</a> — 64-bit LDSMAX	FEAT_LSE
11	0	0	0		0	101	<a href="#">LDSMIN, LDSMINA, LDSMINAL, LDSMINL</a> — 64-bit LDSMIN	FEAT_LSE
11	0	0	0		0	110	<a href="#">LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL</a> — 64-bit LDUMAX	FEAT_LSE
11	0	0	0		0	111	<a href="#">LDUMIN, LDUMINA, LDUMINAL, LDUMINL</a> — 64-bit LDUMIN	FEAT_LSE
11	0	0	0		1	000	<a href="#">SWP, SWPA, SWPAL, SWPL</a> — 64-bit SWP	FEAT_LSE
11	0	0	0		1	010	<a href="#">ST64BV0</a>	FEAT_LS64_ACCDATA
11	0	0	0		1	011	<a href="#">ST64BV</a>	FEAT_LS64_V
11	0	0	0	11111	1	001	<a href="#">ST64B</a>	FEAT_LS64
11	0	0	0	11111	1	101	<a href="#">LD64B</a>	FEAT_LS64
11	0	0	1		0	000	<a href="#">LDADD, LDADDA, LDADDAL, LDADDL</a> — 64-bit LDADDL	FEAT_LSE
11	0	0	1		0	001	<a href="#">LDCLR, LDCLRA, LDCLRAL, LDCLRL</a> — 64-bit LDCLRL	FEAT_LSE
11	0	0	1		0	010	<a href="#">LDEOR, LDEORA, LDEORAL, LDEORL</a> — 64-bit LDEORL	FEAT_LSE
11	0	0	1		0	011	<a href="#">LDSET, LDSETA, LDSETAL, LDSETL</a> — 64-bit LDSETL	FEAT_LSE
11	0	0	1		0	100	<a href="#">LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL</a> — 64-bit LDSMAXL	FEAT_LSE
11	0	0	1		0	101	<a href="#">LDSMIN, LDSMINA, LDSMINAL, LDSMINL</a> — 64-bit LDSMINL	FEAT_LSE
11	0	0	1		0	110	<a href="#">LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL</a> — 64-bit LDUMAXL	FEAT_LSE



size	VR	Decode fields			o3	opc	Instruction Details	Feature
		A	R	Rs				
11	0	0	1		0	111	<a href="#">LDUMIN, LDUMINA, LDUMINAL, LDUMINL</a> — 64-bit LDUMINL	FEAT_LSE
11	0	0	1		1	000	<a href="#">SWP, SWPA, SWPAL, SWPL</a> — 64-bit SWPL	FEAT_LSE
11	0	1	0		0	000	<a href="#">LDADD, LDADDA, LDADDAL, LDADDL</a> — 64-bit LDADDA	FEAT_LSE
11	0	1	0		0	001	<a href="#">LDCLR, LDCLRA, LDCLRAL, LDCLRL</a> — 64-bit LDCLRA	FEAT_LSE
11	0	1	0		0	010	<a href="#">LDEOR, LDEORA, LDEORAL, LDEORL</a> — 64-bit LDEORA	FEAT_LSE
11	0	1	0		0	011	<a href="#">LDSET, LDSETA, LDSETAL, LDSETL</a> — 64-bit LDSETA	FEAT_LSE
11	0	1	0		0	100	<a href="#">LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL</a> — 64-bit LDSMAXA	FEAT_LSE
11	0	1	0		0	101	<a href="#">LDSMIN, LDSMINA, LDSMINAL, LDSMINL</a> — 64-bit LDSMINA	FEAT_LSE
11	0	1	0		0	110	<a href="#">LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL</a> — 64-bit LDUMAXA	FEAT_LSE
11	0	1	0		0	111	<a href="#">LDUMIN, LDUMINA, LDUMINAL, LDUMINL</a> — 64-bit LDUMINA	FEAT_LSE
11	0	1	0		1	000	<a href="#">SWP, SWPA, SWPAL, SWPL</a> — 64-bit SWPA	FEAT_LSE
11	0	1	0		1	100	<a href="#">LDAPR</a> — 64-bit	FEAT_LRCPC
11	0	1	1		0	000	<a href="#">LDADD, LDADDA, LDADDAL, LDADDL</a> — 64-bit LDADDAL	FEAT_LSE
11	0	1	1		0	001	<a href="#">LDCLR, LDCLRA, LDCLRAL, LDCLRL</a> — 64-bit LDCLRAL	FEAT_LSE
11	0	1	1		0	010	<a href="#">LDEOR, LDEORA, LDEORAL, LDEORL</a> — 64-bit LDEORAL	FEAT_LSE
11	0	1	1		0	011	<a href="#">LDSET, LDSETA, LDSETAL, LDSETL</a> — 64-bit LDSETAL	FEAT_LSE
11	0	1	1		0	100	<a href="#">LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL</a> — 64-bit LDSMAXAL	FEAT_LSE
11	0	1	1		0	101	<a href="#">LDSMIN, LDSMINA, LDSMINAL, LDSMINL</a> — 64-bit LDSMINAL	FEAT_LSE
11	0	1	1		0	110	<a href="#">LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL</a> — 64-bit LDUMAXAL	FEAT_LSE
11	0	1	1		0	111	<a href="#">LDUMIN, LDUMINA, LDUMINAL, LDUMINL</a> — 64-bit LDUMINAL	FEAT_LSE
11	0	1	1		1	000	<a href="#">SWP, SWPA, SWPAL, SWPL</a> — 64-bit SWPAL	FEAT_LSE

**Load/store register (register offset)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size	1	1	1	VR	0	0	opc	1			Rm			option	S	1	0			Rn								Rt			

size	VR	Decode fields		Rt	Instruction Details	Feature
		opc	option			
×1	1	1×			UNALLOCATED	-
00	0	00	!= 011		<a href="#">STRB (register)</a> — <a href="#">extended register</a>	-
00	0	00	011		<a href="#">STRB (register)</a> — <a href="#">shifted register</a>	-
00	0	01	!= 011		<a href="#">LDRB (register)</a> — <a href="#">extended register</a>	-

size	VR	Decode fields		Rt	Instruction Details	Feature
		opc	option			
00	0	01	011		<a href="#">LDRB (register) — shifted register</a>	-
00	0	10	!= 011		<a href="#">LDRSB (register) — 64-bit with extended register offset</a>	-
00	0	10	011		<a href="#">LDRSB (register) — 64-bit with shifted register offset</a>	-
00	0	11	!= 011		<a href="#">LDRSB (register) — 32-bit with extended register offset</a>	-
00	0	11	011		<a href="#">LDRSB (register) — 32-bit with shifted register offset</a>	-
00	1	00	!= 011		<a href="#">STR (register, SIMD&amp;FP)</a>	-
00	1	00	011		<a href="#">STR (register, SIMD&amp;FP)</a>	-
00	1	01	!= 011		<a href="#">LDR (register, SIMD&amp;FP)</a>	-
00	1	01	011		<a href="#">LDR (register, SIMD&amp;FP)</a>	-
00	1	10			<a href="#">STR (register, SIMD&amp;FP)</a>	-
00	1	11			<a href="#">LDR (register, SIMD&amp;FP)</a>	-
01	0	00			<a href="#">STRH (register)</a>	-
01	0	01			<a href="#">LDRH (register)</a>	-
01	0	10			<a href="#">LDRSH (register) — 64-bit</a>	-
01	0	11			<a href="#">LDRSH (register) — 32-bit</a>	-
01	1	00			<a href="#">STR (register, SIMD&amp;FP)</a>	-
01	1	01			<a href="#">LDR (register, SIMD&amp;FP)</a>	-
1x	0	11			UNALLOCATED	-
1x	1	1x			UNALLOCATED	-
10	0	00			<a href="#">STR (register) — 32-bit</a>	-
10	0	01			<a href="#">LDR (register) — 32-bit</a>	-
10	0	10			<a href="#">LDRSW (register)</a>	-
10	1	00			<a href="#">STR (register, SIMD&amp;FP)</a>	-
10	1	01			<a href="#">LDR (register, SIMD&amp;FP)</a>	-
11	0	00			<a href="#">STR (register) — 64-bit</a>	-
11	0	01			<a href="#">LDR (register) — 64-bit</a>	-
11	0	10	x0x		UNALLOCATED	-
11	0	10	x1x	!= 11xxx	<a href="#">PRFM (register)</a>	-
11	0	10	x1x	11xxx	<a href="#">RPRFM</a>	FEAT_RPRFM
11	1	00			<a href="#">STR (register, SIMD&amp;FP)</a>	-
11	1	01			<a href="#">LDR (register, SIMD&amp;FP)</a>	-

**Load/store register (pac)**

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	VR	0	0	M	S	1	imm9											W	1	Rn				Rt			

Decode fields				Instruction Details	Feature
size	VR	M	W		
!= 11				UNALLOCATED	-
11	0	0	0	<a href="#">LDRAA, LDRAB</a> — <a href="#">key A, offset</a>	FEAT_PAuth

Decode fields				Instruction Details	Feature
size	VR	M	W		
11	0	0	1	<a href="#">LDRAA, LDRAB</a> — <a href="#">key A, pre-indexed</a>	FEAT_PAuth
11	0	1	0	<a href="#">LDRAA, LDRAB</a> — <a href="#">key B, offset</a>	FEAT_PAuth
11	0	1	1	<a href="#">LDRAA, LDRAB</a> — <a href="#">key B, pre-indexed</a>	FEAT_PAuth
11	1			UNALLOCATED	-

### Load/store register (unsigned immediate)

These instructions are under [Loads and Stores](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	VR	0	1	opc		imm12												Rn				Rt					

Decode fields			Instruction Details
size	VR	opc	
×1	1	1×	UNALLOCATED
00	0	00	<a href="#">STRB (immediate)</a>
00	0	01	<a href="#">LDRB (immediate)</a>
00	0	10	<a href="#">LDRSB (immediate)</a> — 64-bit
00	0	11	<a href="#">LDRSB (immediate)</a> — 32-bit
00	1	00	<a href="#">STR (immediate, SIMD&amp;FP)</a> — 8-bit
00	1	01	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — 8-bit
00	1	10	<a href="#">STR (immediate, SIMD&amp;FP)</a> — 128-bit
00	1	11	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — 128-bit
01	0	00	<a href="#">STRH (immediate)</a>
01	0	01	<a href="#">LDRH (immediate)</a>
01	0	10	<a href="#">LDRSH (immediate)</a> — 64-bit
01	0	11	<a href="#">LDRSH (immediate)</a> — 32-bit
01	1	00	<a href="#">STR (immediate, SIMD&amp;FP)</a> — 16-bit
01	1	01	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — 16-bit
1×	0	11	UNALLOCATED
1×	1	1×	UNALLOCATED
10	0	00	<a href="#">STR (immediate)</a> — 32-bit
10	0	01	<a href="#">LDR (immediate)</a> — 32-bit
10	0	10	<a href="#">LDRSW (immediate)</a>
10	1	00	<a href="#">STR (immediate, SIMD&amp;FP)</a> — 32-bit
10	1	01	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — 32-bit
11	0	00	<a href="#">STR (immediate)</a> — 64-bit
11	0	01	<a href="#">LDR (immediate)</a> — 64-bit
11	0	10	<a href="#">PRFM (immediate)</a>
11	1	00	<a href="#">STR (immediate, SIMD&amp;FP)</a> — 64-bit
11	1	01	<a href="#">LDR (immediate, SIMD&amp;FP)</a> — 64-bit

### Data Processing -- Register

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0			op1			101			op2									op3													

Decode fields

Instruction details

op0	op1	op2	op3	
0	1	0110		<a href="#">Data-processing (2 source)</a>
1	1	0110		<a href="#">Data-processing (1 source)</a>
	0	0xxx		<a href="#">Logical (shifted register)</a>
	0	1xx0		<a href="#">Add/subtract (shifted register)</a>
	0	1xx1		<a href="#">Add/subtract (extended register)</a>
	1	0000	000000	<a href="#">Add/subtract (with carry)</a>
	1	0000	x00001	<a href="#">Rotate right into flags</a>
	1	0000	xx0010	<a href="#">Evaluate into flags</a>
	1	0010	xxxx0x	<a href="#">Conditional compare (register)</a>
	1	0010	xxxx1x	<a href="#">Conditional compare (immediate)</a>
	1	0100		<a href="#">Conditional select</a>
	1	1xxx		<a href="#">Data-processing (3 source)</a>

**Data-processing (2 source)**

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	S	1	1	0	1	0	1	1	0					Rm																

Decode fields			Instruction Details		Feature
sf	S	opcode			
		000001	UNALLOCATED		-
		1xxxxxx	UNALLOCATED		-
	0	00011x	UNALLOCATED		-
	0	001101	UNALLOCATED		-
	0	00111x	UNALLOCATED		-
	0	0111xx	UNALLOCATED		-
	1	00001x	UNALLOCATED		-
	1	0001xx	UNALLOCATED		-
	1	001xxx	UNALLOCATED		-
	1	01xxxx	UNALLOCATED		-
0		000000	UNALLOCATED		-
0	0	000010	<a href="#">UDIV — 32-bit</a>		-
0	0	000011	<a href="#">SDIV — 32-bit</a>		-
0	0	00010x	UNALLOCATED		-
0	0	001000	<a href="#">LSLV — 32-bit</a>		-
0	0	001001	<a href="#">LSRV — 32-bit</a>		-
0	0	001010	<a href="#">ASRV — 32-bit</a>		-
0	0	001011	<a href="#">RORV — 32-bit</a>		-
0	0	001100	UNALLOCATED		-
0	0	010x11	UNALLOCATED		-
0	0	010000	<a href="#">CRC32B, CRC32H, CRC32W, CRC32X — CRC32B</a>		FEAT_CRC32
0	0	010001	<a href="#">CRC32B, CRC32H, CRC32W, CRC32X — CRC32H</a>		FEAT_CRC32
0	0	010010	<a href="#">CRC32B, CRC32H, CRC32W, CRC32X — CRC32W</a>		FEAT_CRC32
0	0	010100	<a href="#">CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CB</a>		FEAT_CRC32
0	0	010101	<a href="#">CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CH</a>		FEAT_CRC32
0	0	010110	<a href="#">CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CW</a>		FEAT_CRC32
0	0	011000	<a href="#">SMAX (register) — 32-bit</a>		FEAT_CSSC

Decode fields			Instruction Details	Feature
sf	S	opcode		
0	0	011001	<a href="#">UMAX (register) — 32-bit</a>	FEAT_CSSC
0	0	011010	<a href="#">SMIN (register) — 32-bit</a>	FEAT_CSSC
0	0	011011	<a href="#">UMIN (register) — 32-bit</a>	FEAT_CSSC
1	0	000000	<a href="#">SUBP</a>	FEAT_MTE
1	0	000010	<a href="#">UDIV — 64-bit</a>	-
1	0	000011	<a href="#">SDIV — 64-bit</a>	-
1	0	000100	<a href="#">IRG</a>	FEAT_MTE
1	0	000101	<a href="#">GMI</a>	FEAT_MTE
1	0	001000	<a href="#">LSLV — 64-bit</a>	-
1	0	001001	<a href="#">LSRV — 64-bit</a>	-
1	0	001010	<a href="#">ASRV — 64-bit</a>	-
1	0	001011	<a href="#">RORV — 64-bit</a>	-
1	0	001100	<a href="#">PACGA</a>	FEAT_PAuth
1	0	010xx0	UNALLOCATED	-
1	0	010x0x	UNALLOCATED	-
1	0	010011	<a href="#">CRC32B, CRC32H, CRC32W, CRC32X — CRC32X</a>	FEAT_CRC32
1	0	010111	<a href="#">CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CX</a>	FEAT_CRC32
1	0	011000	<a href="#">SMAX (register) — 64-bit</a>	FEAT_CSSC
1	0	011001	<a href="#">UMAX (register) — 64-bit</a>	FEAT_CSSC
1	0	011010	<a href="#">SMIN (register) — 64-bit</a>	FEAT_CSSC
1	0	011011	<a href="#">UMIN (register) — 64-bit</a>	FEAT_CSSC
1	1	000000	<a href="#">SUBPS</a>	FEAT_MTE

**Data-processing (1 source)**

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	1	S	1	1	0	1	0	1	1	0	opcode2					opcode					Rn					Rd					

Decode fields				Rn	Instruction Details	Feature
sf	S	opcode2	opcode			
			1xxxxx		UNALLOCATED	-
		xxx1x			UNALLOCATED	-
		xx1xx			UNALLOCATED	-
		x1xxx			UNALLOCATED	-
		1xxxx			UNALLOCATED	-
	0	00000	001001		UNALLOCATED	-
	0	00000	00101x		UNALLOCATED	-
	0	00000	0011xx		UNALLOCATED	-
	0	00000	01xxxx		UNALLOCATED	-
	1				UNALLOCATED	-
0		00001			UNALLOCATED	-
0	0	00000	000000		<a href="#">RBIT — 32-bit</a>	-
0	0	00000	000001		<a href="#">REV16 — 32-bit</a>	-
0	0	00000	000010		<a href="#">REV — 32-bit</a>	-
0	0	00000	000011		UNALLOCATED	-
0	0	00000	000100		<a href="#">CLZ — 32-bit</a>	-
0	0	00000	000101		<a href="#">CLS — 32-bit</a>	-

sf	S	Decode fields		Rn	Instruction Details	Feature
		opcode2	opcode			
0	0	00000	000110		<a href="#">CTZ — 32-bit</a>	FEAT_CSSC
0	0	00000	000111		<a href="#">CNT — 32-bit</a>	FEAT_CSSC
0	0	00000	001000		<a href="#">ABS — 32-bit</a>	FEAT_CSSC
1	0	00000	000000		<a href="#">RBIT — 64-bit</a>	-
1	0	00000	000001		<a href="#">REV16 — 64-bit</a>	-
1	0	00000	000010		<a href="#">REV32</a>	-
1	0	00000	000011		<a href="#">REV — 64-bit</a>	-
1	0	00000	000100		<a href="#">CLZ — 64-bit</a>	-
1	0	00000	000101		<a href="#">CLS — 64-bit</a>	-
1	0	00000	000110		<a href="#">CTZ — 64-bit</a>	FEAT_CSSC
1	0	00000	000111		<a href="#">CNT — 64-bit</a>	FEAT_CSSC
1	0	00000	001000		<a href="#">ABS — 64-bit</a>	FEAT_CSSC
1	0	00001	000000		<a href="#">PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA</a>	FEAT_PAuth
1	0	00001	000001		<a href="#">PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB</a>	FEAT_PAuth
1	0	00001	000010		<a href="#">PACDA, PACDZA — PACDA</a>	FEAT_PAuth
1	0	00001	000011		<a href="#">PACDB, PACDZB — PACDB</a>	FEAT_PAuth
1	0	00001	000100		<a href="#">AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA</a>	FEAT_PAuth
1	0	00001	000101		<a href="#">AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB</a>	FEAT_PAuth
1	0	00001	000110		<a href="#">AUTDA, AUTDZA — AUTDA</a>	FEAT_PAuth
1	0	00001	000111		<a href="#">AUTDB, AUTDZB — AUTDB</a>	FEAT_PAuth
1	0	00001	001000	11111	<a href="#">PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIZA</a>	FEAT_PAuth
1	0	00001	001001	11111	<a href="#">PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIZB</a>	FEAT_PAuth
1	0	00001	001010	11111	<a href="#">PACDA, PACDZA — PACDZA</a>	FEAT_PAuth
1	0	00001	001011	11111	<a href="#">PACDB, PACDZB — PACDZB</a>	FEAT_PAuth
1	0	00001	001100	11111	<a href="#">AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIZA</a>	FEAT_PAuth
1	0	00001	001101	11111	<a href="#">AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIZB</a>	FEAT_PAuth
1	0	00001	001110	11111	<a href="#">AUTDA, AUTDZA — AUTDZA</a>	FEAT_PAuth
1	0	00001	001111	11111	<a href="#">AUTDB, AUTDZB — AUTDZB</a>	FEAT_PAuth
1	0	00001	010000	11111	<a href="#">XPACD, XPACI, XPACLRI — XPACI</a>	FEAT_PAuth
1	0	00001	010001	11111	<a href="#">XPACD, XPACI, XPACLRI — XPACD</a>	FEAT_PAuth
1	0	00001	01001x		UNALLOCATED	-
1	0	00001	0101xx		UNALLOCATED	-
1	0	00001	011xxx		UNALLOCATED	-

**Logical (shifted register)**

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	opc	0	1	0	1	0	shift	N	Rm						imm6						Rn						Rd				

sf	Decode fields		N	imm6	Instruction Details
	opc				
0				1xxxxxx	UNALLOCATED

Decode fields				Instruction Details
sf	opc	N	imm6	
0	00	0		<a href="#">AND (shifted register) — 32-bit</a>
0	00	1		<a href="#">BIC (shifted register) — 32-bit</a>
0	01	0		<a href="#">ORR (shifted register) — 32-bit</a>
0	01	1		<a href="#">ORN (shifted register) — 32-bit</a>
0	10	0		<a href="#">EOR (shifted register) — 32-bit</a>
0	10	1		<a href="#">EON (shifted register) — 32-bit</a>
0	11	0		<a href="#">ANDS (shifted register) — 32-bit</a>
0	11	1		<a href="#">BICS (shifted register) — 32-bit</a>
1	00	0		<a href="#">AND (shifted register) — 64-bit</a>
1	00	1		<a href="#">BIC (shifted register) — 64-bit</a>
1	01	0		<a href="#">ORR (shifted register) — 64-bit</a>
1	01	1		<a href="#">ORN (shifted register) — 64-bit</a>
1	10	0		<a href="#">EOR (shifted register) — 64-bit</a>
1	10	1		<a href="#">EON (shifted register) — 64-bit</a>
1	11	0		<a href="#">ANDS (shifted register) — 64-bit</a>
1	11	1		<a href="#">BICS (shifted register) — 64-bit</a>

**Add/subtract (shifted register)**

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	0	1	0	1	1	shift	0																						

Decode fields					Instruction Details
sf	op	S	shift	imm6	
			11		UNALLOCATED
0				1xxxxxx	UNALLOCATED
0	0	0			<a href="#">ADD (shifted register) — 32-bit</a>
0	0	1			<a href="#">ADDS (shifted register) — 32-bit</a>
0	1	0			<a href="#">SUB (shifted register) — 32-bit</a>
0	1	1			<a href="#">SUBS (shifted register) — 32-bit</a>
1	0	0			<a href="#">ADD (shifted register) — 64-bit</a>
1	0	1			<a href="#">ADDS (shifted register) — 64-bit</a>
1	1	0			<a href="#">SUB (shifted register) — 64-bit</a>
1	1	1			<a href="#">SUBS (shifted register) — 64-bit</a>

**Add/subtract (extended register)**

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	0	1	0	1	1	opt	1																						

Decode fields					Instruction Details
sf	op	S	opt	imm3	
				1x1	UNALLOCATED
				11x	UNALLOCATED
			x1		UNALLOCATED
			1x		UNALLOCATED

Decode fields					Instruction Details
sf	op	S	opt	imm3	
0	0	0	00		<a href="#">ADD (extended register) — 32-bit</a>
0	0	1	00		<a href="#">ADDS (extended register) — 32-bit</a>
0	1	0	00		<a href="#">SUB (extended register) — 32-bit</a>
0	1	1	00		<a href="#">SUBS (extended register) — 32-bit</a>
1	0	0	00		<a href="#">ADD (extended register) — 64-bit</a>
1	0	1	00		<a href="#">ADDS (extended register) — 64-bit</a>
1	1	0	00		<a href="#">SUB (extended register) — 64-bit</a>
1	1	1	00		<a href="#">SUBS (extended register) — 64-bit</a>

**Add/subtract (with carry)**

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0	Rn					Rd				

Decode fields			Instruction Details
sf	op	S	
0	0	0	<a href="#">ADC — 32-bit</a>
0	0	1	<a href="#">ADCS — 32-bit</a>
0	1	0	<a href="#">SBC — 32-bit</a>
0	1	1	<a href="#">SBCS — 32-bit</a>
1	0	0	<a href="#">ADC — 64-bit</a>
1	0	1	<a href="#">ADCS — 64-bit</a>
1	1	0	<a href="#">SBC — 64-bit</a>
1	1	1	<a href="#">SBCS — 64-bit</a>

**Rotate right into flags**

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	1	0	1	0	0	0	0		imm6				0	0	0	0	0	1		Rn				o2	mask			

Decode fields				Instruction Details	Feature
sf	op	S	o2		
0				UNALLOCATED	-
1	0	0		UNALLOCATED	-
1	0	1	0	<a href="#">RMIF</a>	FEAT_FlagM
1	0	1	1	UNALLOCATED	-
1	1			UNALLOCATED	-

**Evaluate into flags**

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	1	0	1	0	0	0	0	opcode2				sz	0	0	1	0	Rn				o3	mask						

Decode fields						Instruction Details	Feature
sf	op	S	opcode2	sz	o3		
0	0	0				UNALLOCATED	-



Decode fields							Instruction Details	Feature
sf	op	S	opcode2	sz	o3	mask		
0	0	1	!= 000000				UNALLOCATED	-
0	0	1	000000		0	!= 1101	UNALLOCATED	-
0	0	1	000000		1		UNALLOCATED	-
0	0	1	000000	0	0	1101	<a href="#">SETF8, SETF16</a> — <a href="#">SETF8</a>	FEAT_FlagM
0	0	1	000000	1	0	1101	<a href="#">SETF8, SETF16</a> — <a href="#">SETF16</a>	FEAT_FlagM
0	1						UNALLOCATED	-
1							UNALLOCATED	-

**Conditional compare (register)**

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	1	0	1	0	0	1	0	Rm					cond					0	o2	Rn					o3	nzcw		

Decode fields					Instruction Details
sf	op	S	o2	o3	
				1	UNALLOCATED
			1		UNALLOCATED
		0			UNALLOCATED
0	0	1	0	0	<a href="#">CCMN (register)</a> — 32-bit
0	1	1	0	0	<a href="#">CCMP (register)</a> — 32-bit
1	0	1	0	0	<a href="#">CCMN (register)</a> — 64-bit
1	1	1	0	0	<a href="#">CCMP (register)</a> — 64-bit

**Conditional compare (immediate)**

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	1	0	1	0	0	1	0	imm5					cond					1	o2	Rn					o3	nzcw		

Decode fields					Instruction Details
sf	op	S	o2	o3	
				1	UNALLOCATED
			1		UNALLOCATED
		0			UNALLOCATED
0	0	1	0	0	<a href="#">CCMN (immediate)</a> — 32-bit
0	1	1	0	0	<a href="#">CCMP (immediate)</a> — 32-bit
1	0	1	0	0	<a href="#">CCMN (immediate)</a> — 64-bit
1	1	1	0	0	<a href="#">CCMP (immediate)</a> — 64-bit

**Conditional select**

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	1	0	1	0	1	0	0	Rm					cond					op2	Rn					Rd				

Decode fields				Instruction Details
sf	op	S	op2	
			1x	UNALLOCATED

Decode fields				Instruction Details
sf	op	S	op2	
		1		UNALLOCATED
0	0	0	00	<a href="#">CSEL — 32-bit</a>
0	0	0	01	<a href="#">CSINC — 32-bit</a>
0	1	0	00	<a href="#">CSINV — 32-bit</a>
0	1	0	01	<a href="#">CSNEG — 32-bit</a>
1	0	0	00	<a href="#">CSEL — 64-bit</a>
1	0	0	01	<a href="#">CSINC — 64-bit</a>
1	1	0	00	<a href="#">CSINV — 64-bit</a>
1	1	0	01	<a href="#">CSNEG — 64-bit</a>

### Data-processing (3 source)

These instructions are under [Data Processing -- Register](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf		op54		1	1	0	1	1	op31			Rm				o0		Ra				Rn				Rd					

Decode fields				Instruction Details
sf	op54	op31	o0	
	00	010	1	UNALLOCATED
	00	011		UNALLOCATED
	00	100		UNALLOCATED
	00	110	1	UNALLOCATED
	00	111		UNALLOCATED
	01			UNALLOCATED
	1x			UNALLOCATED
0	00	000	0	<a href="#">MADD — 32-bit</a>
0	00	000	1	<a href="#">MSUB — 32-bit</a>
0	00	001	0	UNALLOCATED
0	00	001	1	UNALLOCATED
0	00	010	0	UNALLOCATED
0	00	101	0	UNALLOCATED
0	00	101	1	UNALLOCATED
0	00	110	0	UNALLOCATED
1	00	000	0	<a href="#">MADD — 64-bit</a>
1	00	000	1	<a href="#">MSUB — 64-bit</a>
1	00	001	0	<a href="#">SMADDL</a>
1	00	001	1	<a href="#">SMSUBL</a>
1	00	010	0	<a href="#">SMULH</a>
1	00	101	0	<a href="#">UMADDL</a>
1	00	101	1	<a href="#">UMSUBL</a>
1	00	110	0	<a href="#">UMULH</a>

### Data Processing -- Scalar Floating-Point and Advanced SIMD

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0				111		op1		op2		op3																					

Decode fields				Instruction details	Feature
op0	op1	op2	op3		
0000	0x	x101	00xxxxx10	UNALLOCATED	-
0010	0x	x101	00xxxxx10	UNALLOCATED	-
0100	0x	x101	00xxxxx10	<a href="#">Cryptographic AES</a>	-
0101	0x	x0xx	xxx0xxx00	<a href="#">Cryptographic three-register SHA</a>	-
0101	0x	x0xx	xxx0xxx10	UNALLOCATED	-
0101	0x	x101	00xxxxx10	<a href="#">Cryptographic two-register SHA</a>	-
0110	0x	x101	00xxxxx10	UNALLOCATED	-
0111	0x	x0xx	xxx0xxxx0	UNALLOCATED	-
0111	0x	x101	00xxxxx10	UNALLOCATED	-
01x1	00	00xx	xxx0xxxx1	<a href="#">Advanced SIMD scalar copy</a>	-
01x1	01	00xx	xxx0xxxx1	UNALLOCATED	-
01x1	0x	0111	00xxxxx10	UNALLOCATED	-
01x1	0x	10xx	xxx00xxx1	<a href="#">Advanced SIMD scalar three same FP16</a>	-
01x1	0x	10xx	xxx01xxx1	UNALLOCATED	-
01x1	0x	1111	00xxxxx10	<a href="#">Advanced SIMD scalar two-register miscellaneous FP16</a>	-
01x1	0x	x0xx	xxx1xxxx0	UNALLOCATED	-
01x1	0x	x0xx	xxx1xxxx1	<a href="#">Advanced SIMD scalar three same extra</a>	-
01x1	0x	x100	00xxxxx10	<a href="#">Advanced SIMD scalar two-register miscellaneous</a>	-
01x1	0x	x110	00xxxxx10	<a href="#">Advanced SIMD scalar pairwise</a>	-
01x1	0x	x1xx	1xxxxxx10	UNALLOCATED	-
01x1	0x	x1xx	x1xxxxxx10	UNALLOCATED	-
01x1	0x	x1xx	xxxxxxx00	<a href="#">Advanced SIMD scalar three different</a>	-
01x1	0x	x1xx	xxxxxxxxx1	<a href="#">Advanced SIMD scalar three same</a>	-
01x1	10		xxxxxxxxx1	<a href="#">Advanced SIMD scalar shift by immediate</a>	-
01x1	11		xxxxxxxxx1	UNALLOCATED	-
01x1	1x		xxxxxxxxx0	<a href="#">Advanced SIMD scalar x indexed element</a>	-
0x00	0x	x0xx	xxx0xxx00	<a href="#">Advanced SIMD table lookup</a>	-
0x00	0x	x0xx	xxx0xxx10	<a href="#">Advanced SIMD permute</a>	-
0x10	0x	x0xx	xxx0xxxx0	<a href="#">Advanced SIMD extract</a>	-
0xx0	00	00xx	xxx0xxxx1	<a href="#">Advanced SIMD copy</a>	-
0xx0	01	00xx	xxx0xxxx1	UNALLOCATED	-
0xx0	0x	0111	00xxxxx10	UNALLOCATED	-
0xx0	0x	10xx	xxx00xxx1	<a href="#">Advanced SIMD three same (FP16)</a>	-
0xx0	0x	10xx	xxx01xxx1	UNALLOCATED	-
0xx0	0x	1111	00xxxxx10	<a href="#">Advanced SIMD two-register miscellaneous (FP16)</a>	-
0xx0	0x	x0xx	xxx1xxxx0	UNALLOCATED	-
0xx0	0x	x0xx	xxx1xxxx1	<a href="#">Advanced SIMD three-register extension</a>	-
0xx0	0x	x100	00xxxxx10	<a href="#">Advanced SIMD two-register miscellaneous</a>	-
0xx0	0x	x110	00xxxxx10	<a href="#">Advanced SIMD across lanes</a>	-
0xx0	0x	x1xx	1xxxxxx10	UNALLOCATED	-
0xx0	0x	x1xx	x1xxxxxx10	UNALLOCATED	-
0xx0	0x	x1xx	xxxxxxx00	<a href="#">Advanced SIMD three different</a>	-
0xx0	0x	x1xx	xxxxxxxxx1	<a href="#">Advanced SIMD three same</a>	-
0xx0	10	0000	xxxxxxxxx1	<a href="#">Advanced SIMD modified immediate</a>	-
0xx0	10	!= 0000	xxxxxxxxx1	<a href="#">Advanced SIMD shift by immediate</a>	-
0xx0	11		xxxxxxxxx1	UNALLOCATED	-

0xx0	1x		xxxxxxxx0	<a href="#">Advanced SIMD vector x indexed element</a>	-
1100	00	10xx	xxx10xxxx	<a href="#">Cryptographic three-register, imm2</a>	-
1100	00	11xx	xxx1x00xx	<a href="#">Cryptographic three-register SHA 512</a>	-
1100	00		xxx0xxxxx	<a href="#">Cryptographic four-register</a>	-
1100	01	00xx		<a href="#">XAR</a>	FEAT_SHA3
1100	01	1000	0001000xx	<a href="#">Cryptographic two-register SHA 512</a>	-
1xx0	1x			UNALLOCATED	-
x0x1	0x	x0xx		<a href="#">Conversion between floating-point and fixed-point</a>	-
x0x1	0x	x1xx	xxx000000	<a href="#">Conversion between floating-point and integer</a>	-
x0x1	0x	x1xx	xxxx10000	<a href="#">Floating-point data-processing (1 source)</a>	-
x0x1	0x	x1xx	xxxxx1000	<a href="#">Floating-point compare</a>	-
x0x1	0x	x1xx	xxxxxx100	<a href="#">Floating-point immediate</a>	-
x0x1	0x	x1xx	xxxxxxx01	<a href="#">Floating-point conditional compare</a>	-
x0x1	0x	x1xx	xxxxxxx10	<a href="#">Floating-point data-processing (2 source)</a>	-
x0x1	0x	x1xx	xxxxxxx11	<a href="#">Floating-point conditional select</a>	-
x0x1	1x			<a href="#">Floating-point data-processing (3 source)</a>	-

## Cryptographic AES

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	size		1	0	1	0	0	opcode				1	0	Rn				Rd						

Decode fields size	opcode	Instruction Details	Feature
	x1xxx	UNALLOCATED	-
	000xx	UNALLOCATED	-
	1xxxx	UNALLOCATED	-
x1		UNALLOCATED	-
00	00100	<a href="#">AESE</a>	FEAT_AES
00	00101	<a href="#">AESD</a>	FEAT_AES
00	00110	<a href="#">AESMC</a>	FEAT_AES
00	00111	<a href="#">AESIMC</a>	FEAT_AES
1x		UNALLOCATED	-

## Cryptographic three-register SHA

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	0	Rm				0	opcode			0	0	Rn				Rd							

Decode fields size	opcode	Instruction Details	Feature
	111	UNALLOCATED	-
x1		UNALLOCATED	-
00	000	<a href="#">SHA1C</a>	FEAT_SHA1
00	001	<a href="#">SHA1P</a>	FEAT_SHA1
00	010	<a href="#">SHA1M</a>	FEAT_SHA1
00	011	<a href="#">SHA1SU0</a>	FEAT_SHA1
00	100	<a href="#">SHA256H</a>	FEAT_SHA256

Decode fields size	opcode	Instruction Details	Feature
00	101	<a href="#">SHA256H2</a>	FEAT_SHA256
00	110	<a href="#">SHA256SU1</a>	FEAT_SHA256
1x		UNALLOCATED	-

### Cryptographic two-register SHA

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size	1	0	1	0	0	opcode				1	0	Rn				Rd							

Decode fields size	opcode	Instruction Details	Feature
	xx1xx	UNALLOCATED	-
	x1xxx	UNALLOCATED	-
	1xxxx	UNALLOCATED	-
x1		UNALLOCATED	-
00	00000	<a href="#">SHA1H</a>	FEAT_SHA1
00	00001	<a href="#">SHA1SU1</a>	FEAT_SHA1
00	00010	<a href="#">SHA256SU0</a>	FEAT_SHA256
00	00011	UNALLOCATED	-
1x		UNALLOCATED	-

### Advanced SIMD scalar copy

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	op	1	1	1	1	0	0	0	0	imm5				0	imm4				1	Rn				Rd						

Decode fields op	imm4	Instruction Details
0	xxx1	UNALLOCATED
0	xx1x	UNALLOCATED
0	x1xx	UNALLOCATED
0	0000	<a href="#">DUP (element)</a>
0	1xxx	UNALLOCATED
1		UNALLOCATED

### Advanced SIMD scalar three same FP16

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	a	1	0	Rm				0	0	opcode				1	Rn				Rd					

Decode fields U	a	opcode	Instruction Details	Feature
		110	UNALLOCATED	-
	1	011	UNALLOCATED	-
0	0	011	<a href="#">FMULX</a>	FEAT_FP16
0	0	100	<a href="#">FCMEQ (register)</a>	FEAT_FP16

Decode fields			Instruction Details	Feature
U	a	opcode		
0	0	101	UNALLOCATED	-
0	0	111	<a href="#">FRECPS</a>	FEAT_FP16
0	1	100	UNALLOCATED	-
0	1	101	UNALLOCATED	-
0	1	111	<a href="#">FRSQRTS</a>	FEAT_FP16
1	0	011	UNALLOCATED	-
1	0	100	<a href="#">FCMGE (register)</a>	FEAT_FP16
1	0	101	<a href="#">FACGE</a>	FEAT_FP16
1	0	111	UNALLOCATED	-
1	1	010	<a href="#">FABD</a>	FEAT_FP16
1	1	100	<a href="#">FCMGT (register)</a>	FEAT_FP16
1	1	101	<a href="#">FACGT</a>	FEAT_FP16
1	1	111	UNALLOCATED	-

### Advanced SIMD scalar two-register miscellaneous FP16

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	a	1	1	1	1	0	0	opcode				1		0	Rn				Rd					

Decode fields			Instruction Details	Feature
U	a	opcode		
		00xxx	UNALLOCATED	-
		010xx	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		1100x	UNALLOCATED	-
		11110	UNALLOCATED	-
	0	011xx	UNALLOCATED	-
	0	11111	UNALLOCATED	-
	1	01111	UNALLOCATED	-
	1	11100	UNALLOCATED	-
0	0	11010	<a href="#">FCVTNS (vector)</a>	FEAT_FP16
0	0	11011	<a href="#">FCVTMS (vector)</a>	FEAT_FP16
0	0	11100	<a href="#">FCVTAS (vector)</a>	FEAT_FP16
0	0	11101	<a href="#">SCVTF (vector, integer)</a>	FEAT_FP16
0	1	01100	<a href="#">FCMGT (zero)</a>	FEAT_FP16
0	1	01101	<a href="#">FCMEQ (zero)</a>	FEAT_FP16
0	1	01110	<a href="#">FCMLT (zero)</a>	FEAT_FP16
0	1	11010	<a href="#">FCVTPS (vector)</a>	FEAT_FP16
0	1	11011	<a href="#">FCVTZS (vector, integer)</a>	FEAT_FP16
0	1	11101	<a href="#">FRECPE</a>	FEAT_FP16
0	1	11111	<a href="#">FRECPX</a>	FEAT_FP16
1	0	11010	<a href="#">FCVTNU (vector)</a>	FEAT_FP16
1	0	11011	<a href="#">FCVTMU (vector)</a>	FEAT_FP16
1	0	11100	<a href="#">FCVTAU (vector)</a>	FEAT_FP16
1	0	11101	<a href="#">UCVTF (vector, integer)</a>	FEAT_FP16
1	1	01100	<a href="#">FCMGE (zero)</a>	FEAT_FP16
1	1	01101	<a href="#">FCMLE (zero)</a>	FEAT_FP16

Decode fields			Instruction Details	Feature
U	a	opcode		
1	1	01110	UNALLOCATED	-
1	1	11010	<a href="#">FCVTPU (vector)</a>	FEAT_FP16
1	1	11011	<a href="#">FCVTZU (vector, integer)</a>	FEAT_FP16
1	1	11101	<a href="#">FRSQRT</a>	FEAT_FP16
1	1	11111	UNALLOCATED	-

### Advanced SIMD scalar three same extra

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	size	0				Rm			1		opcode	1												Rd

Decode fields		Instruction Details	Feature
U	opcode		
	001x	UNALLOCATED	-
	01xx	UNALLOCATED	-
	1xxx	UNALLOCATED	-
0	0000	UNALLOCATED	-
0	0001	UNALLOCATED	-
1	0000	<a href="#">SQRDMMLAH (vector)</a>	FEAT_RDM
1	0001	<a href="#">SQRDMMLSH (vector)</a>	FEAT_RDM

### Advanced SIMD scalar two-register miscellaneous

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	size	1	0	0	0	0					opcode		1	0										Rd

Decode fields			Instruction Details
U	size	opcode	
		0000x	UNALLOCATED
		00010	UNALLOCATED
		0010x	UNALLOCATED
		00110	UNALLOCATED
		01111	UNALLOCATED
		1000x	UNALLOCATED
		10011	UNALLOCATED
		10101	UNALLOCATED
		10111	UNALLOCATED
		1100x	UNALLOCATED
		11110	UNALLOCATED
	0x	011xx	UNALLOCATED
	0x	11111	UNALLOCATED
	1x	10110	UNALLOCATED
	1x	11100	UNALLOCATED
0		00011	<a href="#">SUQADD</a>
0		00111	<a href="#">SQABS</a>
0		01000	<a href="#">CMGT (zero)</a>

Decode fields			Instruction Details
U	size	opcode	
0		01001	<a href="#">CMEQ (zero)</a>
0		01010	<a href="#">CMLT (zero)</a>
0		01011	<a href="#">ABS</a>
0		10010	UNALLOCATED
0		10100	<a href="#">SQXTN, SQXTN2</a>
0	0x	10110	UNALLOCATED
0	0x	11010	<a href="#">FCVTNS (vector)</a>
0	0x	11011	<a href="#">FCVTMS (vector)</a>
0	0x	11100	<a href="#">FCVTAS (vector)</a>
0	0x	11101	<a href="#">SCVTF (vector, integer)</a>
0	1x	01100	<a href="#">FCMGT (zero)</a>
0	1x	01101	<a href="#">FCMEQ (zero)</a>
0	1x	01110	<a href="#">FCMLT (zero)</a>
0	1x	11010	<a href="#">FCVTPS (vector)</a>
0	1x	11011	<a href="#">FCVTZS (vector, integer)</a>
0	1x	11101	<a href="#">FRECPE</a>
0	1x	11111	<a href="#">FRECPX</a>
1		00011	<a href="#">USQADD</a>
1		00111	<a href="#">SQNEG</a>
1		01000	<a href="#">CMGE (zero)</a>
1		01001	<a href="#">CMLE (zero)</a>
1		01010	UNALLOCATED
1		01011	<a href="#">NEG (vector)</a>
1		10010	<a href="#">SQXTUN, SQXTUN2</a>
1		10100	<a href="#">UQXTN, UQXTN2</a>
1	0x	10110	<a href="#">FCVTXN, FCVTXN2</a>
1	0x	11010	<a href="#">FCVTNU (vector)</a>
1	0x	11011	<a href="#">FCVTMU (vector)</a>
1	0x	11100	<a href="#">FCVTAU (vector)</a>
1	0x	11101	<a href="#">UCVTF (vector, integer)</a>
1	1x	01100	<a href="#">FCMGE (zero)</a>
1	1x	01101	<a href="#">FCMLE (zero)</a>
1	1x	01110	UNALLOCATED
1	1x	11010	<a href="#">FCVTPU (vector)</a>
1	1x	11011	<a href="#">FCVTZU (vector, integer)</a>
1	1x	11101	<a href="#">FRSQRT</a>
1	1x	11111	UNALLOCATED

### Advanced SIMD scalar pairwise

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	size	1	1	0	0	0	opcode			1	0	Rn				Rd								

Decode fields			Instruction Details	Feature
U	size	opcode		
		00xxx	UNALLOCATED	-
		010xx	UNALLOCATED	-



Decode fields			Instruction Details	Feature
U	size	opcode		
		01110	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		1100x	UNALLOCATED	-
		11010	UNALLOCATED	-
		111xx	UNALLOCATED	-
	1x	01101	UNALLOCATED	-
0		11011	<a href="#">ADDP (scalar)</a>	-
0	0x	01100	<a href="#">FMAXNMP (scalar) — half-precision</a>	FEAT_FP16
0	0x	01101	<a href="#">FADDP (scalar) — half-precision</a>	FEAT_FP16
0	0x	01111	<a href="#">FMAXP (scalar) — half-precision</a>	FEAT_FP16
0	1x	01100	<a href="#">FMINNMP (scalar) — half-precision</a>	FEAT_FP16
0	1x	01111	<a href="#">FMINP (scalar) — half-precision</a>	FEAT_FP16
1		11011	UNALLOCATED	-
1	0x	01100	<a href="#">FMAXNMP (scalar) — single-precision and double-precision</a>	-
1	0x	01101	<a href="#">FADDP (scalar) — single-precision and double-precision</a>	-
1	0x	01111	<a href="#">FMAXP (scalar) — single-precision and double-precision</a>	-
1	1x	01100	<a href="#">FMINNMP (scalar) — single-precision and double-precision</a>	-
1	1x	01111	<a href="#">FMINP (scalar) — single-precision and double-precision</a>	-

**Advanced SIMD scalar three different**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	size	1	Rm			opcode			0	0	Rn			Rd										

Decode fields		Instruction Details
U	opcode	
	00xx	UNALLOCATED
	01xx	UNALLOCATED
	1000	UNALLOCATED
	1010	UNALLOCATED
	1100	UNALLOCATED
	111x	UNALLOCATED
0	1001	<a href="#">SQDMLAL, SQDMLAL2 (vector)</a>
0	1011	<a href="#">SQDMLSL, SQDMLSL2 (vector)</a>
0	1101	<a href="#">SQDMULL, SQDMULL2 (vector)</a>
1	1001	UNALLOCATED
1	1011	UNALLOCATED
1	1101	UNALLOCATED

**Advanced SIMD scalar three same**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	size	1	Rm				opcode				1	Rn				Rd								

U	Decode fields		Instruction Details
	size	opcode	
		00000	UNALLOCATED
		0001x	UNALLOCATED
		00100	UNALLOCATED
		011xx	UNALLOCATED
		1001x	UNALLOCATED
	1x	11011	UNALLOCATED
0		00001	<a href="#">SQADD</a>
0		00101	<a href="#">SQSUB</a>
0		00110	<a href="#">CMGT (register)</a>
0		00111	<a href="#">CMGE (register)</a>
0		01000	<a href="#">SSHL</a>
0		01001	<a href="#">SQSHL (register)</a>
0		01010	<a href="#">SRSHL</a>
0		01011	<a href="#">SQRSHL</a>
0		10000	<a href="#">ADD (vector)</a>
0		10001	<a href="#">CMTST</a>
0		10100	UNALLOCATED
0		10101	UNALLOCATED
0		10110	<a href="#">SQDMULH (vector)</a>
0		10111	UNALLOCATED
0	0x	11000	UNALLOCATED
0	0x	11001	UNALLOCATED
0	0x	11010	UNALLOCATED
0	0x	11011	<a href="#">FMULX</a>
0	0x	11100	<a href="#">FCMEQ (register)</a>
0	0x	11101	UNALLOCATED
0	0x	11110	UNALLOCATED
0	0x	11111	<a href="#">FRECPS</a>
0	1x	11000	UNALLOCATED
0	1x	11001	UNALLOCATED
0	1x	11010	UNALLOCATED
0	1x	11100	UNALLOCATED
0	1x	11101	UNALLOCATED
0	1x	11110	UNALLOCATED
0	1x	11111	<a href="#">FRSQRTS</a>
1		00001	<a href="#">UQADD</a>
1		00101	<a href="#">UQSUB</a>
1		00110	<a href="#">CMHI (register)</a>
1		00111	<a href="#">CMHS (register)</a>
1		01000	<a href="#">USHL</a>
1		01001	<a href="#">UQSHL (register)</a>
1		01010	<a href="#">URSHL</a>
1		01011	<a href="#">UQRSHL</a>
1		10000	<a href="#">SUB (vector)</a>
1		10001	<a href="#">CMEQ (register)</a>
1		10100	UNALLOCATED
1		10101	UNALLOCATED

Decode fields			Instruction Details
U	size	opcode	
1		10110	<a href="#">SQRDMULH (vector)</a>
1		10111	UNALLOCATED
1	0x	11000	UNALLOCATED
1	0x	11001	UNALLOCATED
1	0x	11010	UNALLOCATED
1	0x	11011	UNALLOCATED
1	0x	11100	<a href="#">FCMGE (register)</a>
1	0x	11101	<a href="#">FACGE</a>
1	0x	11110	UNALLOCATED
1	0x	11111	UNALLOCATED
1	1x	11000	UNALLOCATED
1	1x	11001	UNALLOCATED
1	1x	11010	<a href="#">FABD</a>
1	1x	11100	<a href="#">FCMGT (register)</a>
1	1x	11101	<a href="#">FACGT</a>
1	1x	11110	UNALLOCATED
1	1x	11111	UNALLOCATED

**Advanced SIMD scalar shift by immediate**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	1	0	immh				immb		opcode						1	Rn				Rd					

Decode fields			Instruction Details
U	immh	opcode	
	!= 0000	00001	UNALLOCATED
	!= 0000	00011	UNALLOCATED
	!= 0000	00101	UNALLOCATED
	!= 0000	00111	UNALLOCATED
	!= 0000	01001	UNALLOCATED
	!= 0000	01011	UNALLOCATED
	!= 0000	01101	UNALLOCATED
	!= 0000	01111	UNALLOCATED
	!= 0000	101xx	UNALLOCATED
	!= 0000	110xx	UNALLOCATED
	!= 0000	11101	UNALLOCATED
	!= 0000	11110	UNALLOCATED
	0000		UNALLOCATED
0	!= 0000	00000	<a href="#">SSHR</a>
0	!= 0000	00010	<a href="#">SSRA</a>
0	!= 0000	00100	<a href="#">SRSHR</a>
0	!= 0000	00110	<a href="#">SRSRA</a>
0	!= 0000	01000	UNALLOCATED
0	!= 0000	01010	<a href="#">SHL</a>
0	!= 0000	01100	UNALLOCATED
0	!= 0000	01110	<a href="#">SQSHL (immediate)</a>
0	!= 0000	10000	UNALLOCATED

Decode fields			Instruction Details
U	immh	opcode	
0	!= 0000	10001	UNALLOCATED
0	!= 0000	10010	<a href="#">SQSHRN, SQSHRN2</a>
0	!= 0000	10011	<a href="#">SQRSHRN, SQRSHRN2</a>
0	!= 0000	11100	<a href="#">SCVTF (vector, fixed-point)</a>
0	!= 0000	11111	<a href="#">FCVTZS (vector, fixed-point)</a>
1	!= 0000	00000	<a href="#">USHR</a>
1	!= 0000	00010	<a href="#">USRA</a>
1	!= 0000	00100	<a href="#">URSHR</a>
1	!= 0000	00110	<a href="#">URSRA</a>
1	!= 0000	01000	<a href="#">SRI</a>
1	!= 0000	01010	<a href="#">SLI</a>
1	!= 0000	01100	<a href="#">SQSHLU</a>
1	!= 0000	01110	<a href="#">UQSHL (immediate)</a>
1	!= 0000	10000	<a href="#">SQSHRUN, SQSHRUN2</a>
1	!= 0000	10001	<a href="#">SQRSHRUN, SQRSHRUN2</a>
1	!= 0000	10010	<a href="#">UQSHRN, UQSHRN2</a>
1	!= 0000	10011	<a href="#">UQRSHRN, UQRSHRN2</a>
1	!= 0000	11100	<a href="#">UCVTF (vector, fixed-point)</a>
1	!= 0000	11111	<a href="#">FCVTZU (vector, fixed-point)</a>

**Advanced SIMD scalar x indexed element**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	1	size	L	M	Rm				opcode				H	0	Rn				Rd						

Decode fields			Instruction Details	Feature
U	size	opcode		
		0000	UNALLOCATED	-
		0010	UNALLOCATED	-
		0100	UNALLOCATED	-
		0110	UNALLOCATED	-
		1000	UNALLOCATED	-
		1010	UNALLOCATED	-
		1110	UNALLOCATED	-
	01	0001	UNALLOCATED	-
	01	0101	UNALLOCATED	-
	01	1001	UNALLOCATED	-
0		0011	<a href="#">SQDMLAL, SQDMLAL2 (by element)</a>	-
0		0111	<a href="#">SQDMLSL, SQDMLSL2 (by element)</a>	-
0		1011	<a href="#">SQDMULL, SQDMULL2 (by element)</a>	-
0		1100	<a href="#">SQDMULH (by element)</a>	-
0		1101	<a href="#">SQRDMULH (by element)</a>	-
0		1111	UNALLOCATED	-
0	00	0001	<a href="#">FMLA (by element) — half-precision</a>	FEAT_FP16
0	00	0101	<a href="#">FMLS (by element) — half-precision</a>	FEAT_FP16
0	00	1001	<a href="#">FMUL (by element) — half-precision</a>	FEAT_FP16
0	1x	0001	<a href="#">FMLA (by element) — single-precision and double-precision</a>	-

Decode fields			Instruction Details	Feature
U	size	opcode		
0	1x	0101	<a href="#">FMLS (by element) — single-precision and double-precision</a>	-
0	1x	1001	<a href="#">FMUL (by element) — single-precision and double-precision</a>	-
1		0011	UNALLOCATED	-
1		0111	UNALLOCATED	-
1		1011	UNALLOCATED	-
1		1100	UNALLOCATED	-
1		1101	<a href="#">SQRDMLAH (by element)</a>	FEAT_RDM
1		1111	<a href="#">SQRDMLSH (by element)</a>	FEAT_RDM
1	00	0001	UNALLOCATED	-
1	00	0101	UNALLOCATED	-
1	00	1001	<a href="#">FMULX (by element) — half-precision</a>	FEAT_FP16
1	1x	0001	UNALLOCATED	-
1	1x	0101	UNALLOCATED	-
1	1x	1001	<a href="#">FMULX (by element) — single-precision and double-precision</a>	-

### Advanced SIMD table lookup

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	op2	0	Rm				0	len	op	0	0	Rn				Rd								

Decode fields			Instruction Details
op2	len	op	
x1			UNALLOCATED
00	00	0	<a href="#">TBL — single register table</a>
00	00	1	<a href="#">TBX — single register table</a>
00	01	0	<a href="#">TBL — two register table</a>
00	01	1	<a href="#">TBX — two register table</a>
00	10	0	<a href="#">TBL — three register table</a>
00	10	1	<a href="#">TBX — three register table</a>
00	11	0	<a href="#">TBL — four register table</a>
00	11	1	<a href="#">TBX — four register table</a>
1x			UNALLOCATED

### Advanced SIMD permute

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size	0	Rm				0	opcode		1	0	Rn				Rd								

Decode fields	Instruction Details
opcode	
000	UNALLOCATED
001	<a href="#">UZP1</a>
010	<a href="#">TRN1</a>
011	<a href="#">ZIP1</a>
100	UNALLOCATED
101	<a href="#">UZP2</a>

Decode fields opcode	Instruction Details
110	<a href="#">TRN2</a>
111	<a href="#">ZIP2</a>

### Advanced SIMD extract

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	op2	0	Rm				0	imm4				0	Rn				Rd							

Decode fields op2	Instruction Details
×1	UNALLOCATED
00	<a href="#">EXT</a>
1×	UNALLOCATED

### Advanced SIMD copy

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	op	0	1	1	1	0	0	0	0	imm5				0	imm4				1	Rn				Rd						

Q	op	Decode fields imm5	imm4	Instruction Details
		×0000		UNALLOCATED
	0		0000	<a href="#">DUP (element)</a>
	0		0001	<a href="#">DUP (general)</a>
	0		0010	UNALLOCATED
	0		0100	UNALLOCATED
	0		0110	UNALLOCATED
	0		1×××	UNALLOCATED
0	0		0011	UNALLOCATED
0	0		0101	<a href="#">SMOV</a>
0	0		0111	<a href="#">UMOV</a>
0	1			UNALLOCATED
1	0		0011	<a href="#">INS (general)</a>
1	0		0101	<a href="#">SMOV</a>
1	0	×1000	0111	<a href="#">UMOV</a>
1	1			<a href="#">INS (element)</a>

### Advanced SIMD three same (FP16)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	a	1	0	Rm				0	0	opcode				1	Rn				Rd					

U	a	Decode fields opcode	Instruction Details	Feature
0	0	000	<a href="#">FMAXNM (vector)</a>	FEAT_FP16
0	0	001	<a href="#">FMLA (vector)</a>	FEAT_FP16

Decode fields			Instruction Details	Feature
U	a	opcode		
0	0	010	<a href="#">FADD (vector)</a>	FEAT_FP16
0	0	011	<a href="#">FMULX</a>	FEAT_FP16
0	0	100	<a href="#">FCMEQ (register)</a>	FEAT_FP16
0	0	101	UNALLOCATED	-
0	0	110	<a href="#">FMAX (vector)</a>	FEAT_FP16
0	0	111	<a href="#">FRECPS</a>	FEAT_FP16
0	1	000	<a href="#">FMINNM (vector)</a>	FEAT_FP16
0	1	001	<a href="#">FMLS (vector)</a>	FEAT_FP16
0	1	010	<a href="#">FSUB (vector)</a>	FEAT_FP16
0	1	011	UNALLOCATED	-
0	1	100	UNALLOCATED	-
0	1	101	UNALLOCATED	-
0	1	110	<a href="#">FMIN (vector)</a>	FEAT_FP16
0	1	111	<a href="#">FRSQRTS</a>	FEAT_FP16
1	0	000	<a href="#">FMAXNMP (vector)</a>	FEAT_FP16
1	0	001	UNALLOCATED	-
1	0	010	<a href="#">FADDP (vector)</a>	FEAT_FP16
1	0	011	<a href="#">FMUL (vector)</a>	FEAT_FP16
1	0	100	<a href="#">FCMGE (register)</a>	FEAT_FP16
1	0	101	<a href="#">FACGE</a>	FEAT_FP16
1	0	110	<a href="#">FMAXP (vector)</a>	FEAT_FP16
1	0	111	<a href="#">FDIV (vector)</a>	FEAT_FP16
1	1	000	<a href="#">FMINNMP (vector)</a>	FEAT_FP16
1	1	001	UNALLOCATED	-
1	1	010	<a href="#">FABD</a>	FEAT_FP16
1	1	011	UNALLOCATED	-
1	1	100	<a href="#">FCMGT (register)</a>	FEAT_FP16
1	1	101	<a href="#">FACGT</a>	FEAT_FP16
1	1	110	<a href="#">FMINP (vector)</a>	FEAT_FP16
1	1	111	UNALLOCATED	-

**Advanced SIMD two-register miscellaneous (FP16)**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	a	1	1	1	1	0	0	opcode				1	0	Rn				Rd						

Decode fields			Instruction Details	Feature
U	a	opcode		
		00xxx	UNALLOCATED	-
		010xx	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		11110	UNALLOCATED	-
	0	011xx	UNALLOCATED	-
	0	11111	UNALLOCATED	-
	1	11100	UNALLOCATED	-
0	0	11000	<a href="#">FRINTN (vector)</a>	FEAT_FP16
0	0	11001	<a href="#">FRINTM (vector)</a>	FEAT_FP16

Decode fields			Instruction Details	Feature
U	a	opcode		
0	0	11010	<a href="#">FCVTNS (vector)</a>	FEAT_FP16
0	0	11011	<a href="#">FCVTMS (vector)</a>	FEAT_FP16
0	0	11100	<a href="#">FCVTAS (vector)</a>	FEAT_FP16
0	0	11101	<a href="#">SCVTF (vector, integer)</a>	FEAT_FP16
0	1	01100	<a href="#">FCMGT (zero)</a>	FEAT_FP16
0	1	01101	<a href="#">FCMEQ (zero)</a>	FEAT_FP16
0	1	01110	<a href="#">FCMLT (zero)</a>	FEAT_FP16
0	1	01111	<a href="#">FABS (vector)</a>	FEAT_FP16
0	1	11000	<a href="#">FRINTP (vector)</a>	FEAT_FP16
0	1	11001	<a href="#">FRINTZ (vector)</a>	FEAT_FP16
0	1	11010	<a href="#">FCVTPS (vector)</a>	FEAT_FP16
0	1	11011	<a href="#">FCVTZS (vector, integer)</a>	FEAT_FP16
0	1	11101	<a href="#">FRECPE</a>	FEAT_FP16
0	1	11111	UNALLOCATED	-
1	0	11000	<a href="#">FRINTA (vector)</a>	FEAT_FP16
1	0	11001	<a href="#">FRINTX (vector)</a>	FEAT_FP16
1	0	11010	<a href="#">FCVTNU (vector)</a>	FEAT_FP16
1	0	11011	<a href="#">FCVTMU (vector)</a>	FEAT_FP16
1	0	11100	<a href="#">FCVTAU (vector)</a>	FEAT_FP16
1	0	11101	<a href="#">UCVTF (vector, integer)</a>	FEAT_FP16
1	1	01100	<a href="#">FCMGE (zero)</a>	FEAT_FP16
1	1	01101	<a href="#">FCMLE (zero)</a>	FEAT_FP16
1	1	01110	UNALLOCATED	-
1	1	01111	<a href="#">FNEG (vector)</a>	FEAT_FP16
1	1	11000	UNALLOCATED	-
1	1	11001	<a href="#">FRINTI (vector)</a>	FEAT_FP16
1	1	11010	<a href="#">FCVTPU (vector)</a>	FEAT_FP16
1	1	11011	<a href="#">FCVTZU (vector, integer)</a>	FEAT_FP16
1	1	11101	<a href="#">FRSQRT</a>	FEAT_FP16
1	1	11111	<a href="#">FSQRT (vector)</a>	FEAT_FP16

### Advanced SIMD three-register extension

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	size	0		Rm		1		opcode	1		Rn											Rd		

Decode fields				Instruction Details	Feature
Q	U	size	opcode		
		0x	0011	UNALLOCATED	-
		11	0011	UNALLOCATED	-
	0		0000	UNALLOCATED	-
	0		0001	UNALLOCATED	-
	0		0010	<a href="#">SDOT (vector)</a>	FEAT_DotProd
	0		1xxx	UNALLOCATED	-
	0	10	0011	<a href="#">USDOT (vector)</a>	FEAT_I8MM
	1		0000	<a href="#">SQRDMLAH (vector)</a>	FEAT_RDM
	1		0001	<a href="#">SQRDMLSH (vector)</a>	FEAT_RDM



Q	Decode fields			Instruction Details	Feature
	U	size	opcode		
	1		0010	<a href="#">UDOT (vector)</a>	FEAT_DotProd
	1		10xx	<a href="#">FCMLA</a>	FEAT_FCMA
	1		11x0	<a href="#">FCADD</a>	FEAT_FCMA
	1	00	1101	UNALLOCATED	-
	1	00	1111	UNALLOCATED	-
	1	01	1111	<a href="#">BFDOT (vector)</a>	FEAT_BF16
	1	1x	1101	UNALLOCATED	-
	1	10	0011	UNALLOCATED	-
	1	10	1111	UNALLOCATED	-
	1	11	1111	<a href="#">BFMLALB, BFMLALT (vector)</a>	FEAT_BF16
0			01xx	UNALLOCATED	-
0	1	01	1101	UNALLOCATED	-
1		0x	01xx	UNALLOCATED	-
1		1x	011x	UNALLOCATED	-
1	0	10	0100	<a href="#">SMMLA (vector)</a>	FEAT_I8MM
1	0	10	0101	<a href="#">USMMLA (vector)</a>	FEAT_I8MM
1	1	01	1101	<a href="#">BFMMLA</a>	FEAT_BF16
1	1	10	0100	<a href="#">UMMLA (vector)</a>	FEAT_I8MM
1	1	10	0101	UNALLOCATED	-

**Advanced SIMD two-register miscellaneous**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	size	1	0	0	0	0	opcode	1	0	Rn														Rd

U	Decode fields		Instruction Details	Feature
	size	opcode		
		1000x	UNALLOCATED	-
		10101	UNALLOCATED	-
	0x	011xx	UNALLOCATED	-
	1x	10111	UNALLOCATED	-
	1x	11110	UNALLOCATED	-
	11	10110	UNALLOCATED	-
0		00000	<a href="#">REV64</a>	-
0		00001	<a href="#">REV16 (vector)</a>	-
0		00010	<a href="#">SADDLP</a>	-
0		00011	<a href="#">SUQADD</a>	-
0		00100	<a href="#">CLS (vector)</a>	-
0		00101	<a href="#">CNT</a>	-
0		00110	<a href="#">SADALP</a>	-
0		00111	<a href="#">SQABS</a>	-
0		01000	<a href="#">CMGT (zero)</a>	-
0		01001	<a href="#">CMEQ (zero)</a>	-
0		01010	<a href="#">CMLT (zero)</a>	-
0		01011	<a href="#">ABS</a>	-
0		10010	<a href="#">XTN, XTN2</a>	-
0		10011	UNALLOCATED	-

U	Decode fields size opcode	Instruction Details	Feature
0		10100 <a href="#">SQXTN, SQXTN2</a>	-
0	0x	10110 <a href="#">FCVTN, FCVTN2</a>	-
0	0x	10111 <a href="#">FCVTL, FCVTL2</a>	-
0	0x	11000 <a href="#">FRINTN (vector)</a>	-
0	0x	11001 <a href="#">FRINTM (vector)</a>	-
0	0x	11010 <a href="#">FCVTNS (vector)</a>	-
0	0x	11011 <a href="#">FCVTMS (vector)</a>	-
0	0x	11100 <a href="#">FCVTAS (vector)</a>	-
0	0x	11101 <a href="#">SCVTF (vector, integer)</a>	-
0	0x	11110 <a href="#">FRINT32Z (vector)</a>	FEAT_FRINTTS
0	0x	11111 <a href="#">FRINT64Z (vector)</a>	FEAT_FRINTTS
0	1x	01100 <a href="#">FCMGT (zero)</a>	-
0	1x	01101 <a href="#">FCMEQ (zero)</a>	-
0	1x	01110 <a href="#">FCMLT (zero)</a>	-
0	1x	01111 <a href="#">FABS (vector)</a>	-
0	1x	11000 <a href="#">FRINTP (vector)</a>	-
0	1x	11001 <a href="#">FRINTZ (vector)</a>	-
0	1x	11010 <a href="#">FCVTPS (vector)</a>	-
0	1x	11011 <a href="#">FCVTZS (vector, integer)</a>	-
0	1x	11100 <a href="#">URECPE</a>	-
0	1x	11101 <a href="#">FRECPE</a>	-
0	1x	11111 UNALLOCATED	-
0	10	10110 <a href="#">BFCVTN, BFCVTN2</a>	FEAT_BF16
1		00000 <a href="#">REV32 (vector)</a>	-
1		00001 UNALLOCATED	-
1		00010 <a href="#">UADDLP</a>	-
1		00011 <a href="#">USQADD</a>	-
1		00100 <a href="#">CLZ (vector)</a>	-
1		00110 <a href="#">UADALP</a>	-
1		00111 <a href="#">SQNEG</a>	-
1		01000 <a href="#">CMGE (zero)</a>	-
1		01001 <a href="#">CMLE (zero)</a>	-
1		01010 UNALLOCATED	-
1		01011 <a href="#">NEG (vector)</a>	-
1		10010 <a href="#">SQXTUN, SQXTUN2</a>	-
1		10011 <a href="#">SHLL, SHLL2</a>	-
1		10100 <a href="#">UQXTN, UQXTN2</a>	-
1	0x	10110 <a href="#">FCVTXN, FCVTXN2</a>	-
1	0x	10111 UNALLOCATED	-
1	0x	11000 <a href="#">FRINTA (vector)</a>	-
1	0x	11001 <a href="#">FRINTX (vector)</a>	-
1	0x	11010 <a href="#">FCVTNU (vector)</a>	-
1	0x	11011 <a href="#">FCVTMU (vector)</a>	-
1	0x	11100 <a href="#">FCVTAU (vector)</a>	-
1	0x	11101 <a href="#">UCVTF (vector, integer)</a>	-
1	0x	11110 <a href="#">FRINT32X (vector)</a>	FEAT_FRINTTS
1	0x	11111 <a href="#">FRINT64X (vector)</a>	FEAT_FRINTTS

Decode fields			Instruction Details	Feature
U	size	opcode		
1	00	00101	<a href="#">NOT</a>	-
1	01	00101	<a href="#">RBIT (vector)</a>	-
1	1x	00101	UNALLOCATED	-
1	1x	01100	<a href="#">FCMGE (zero)</a>	-
1	1x	01101	<a href="#">FCMLE (zero)</a>	-
1	1x	01110	UNALLOCATED	-
1	1x	01111	<a href="#">FNEG (vector)</a>	-
1	1x	11000	UNALLOCATED	-
1	1x	11001	<a href="#">FRINTI (vector)</a>	-
1	1x	11010	<a href="#">FCVTPU (vector)</a>	-
1	1x	11011	<a href="#">FCVTZU (vector, integer)</a>	-
1	1x	11100	<a href="#">URSQRTE</a>	-
1	1x	11101	<a href="#">FRSQRTE</a>	-
1	1x	11111	<a href="#">FSQRT (vector)</a>	-
1	10	10110	UNALLOCATED	-

### Advanced SIMD across lanes

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	size	1	1	0	0	0	opcode			1		0	Rn					Rd						

Decode fields			Instruction Details	Feature
U	size	opcode		
		0000x	UNALLOCATED	-
		00010	UNALLOCATED	-
		001xx	UNALLOCATED	-
		0100x	UNALLOCATED	-
		01011	UNALLOCATED	-
		01101	UNALLOCATED	-
		01110	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		1100x	UNALLOCATED	-
		111xx	UNALLOCATED	-
0		00011	<a href="#">SADDLV</a>	-
0		01010	<a href="#">SMAXV</a>	-
0		11010	<a href="#">SMINV</a>	-
0		11011	<a href="#">ADDV</a>	-
0	00	01100	<a href="#">FMAXNMV — half-precision</a>	FEAT_FP16
0	00	01111	<a href="#">FMAXV — half-precision</a>	FEAT_FP16
0	01	01100	UNALLOCATED	-
0	01	01111	UNALLOCATED	-
0	10	01100	<a href="#">FMINNMV — half-precision</a>	FEAT_FP16
0	10	01111	<a href="#">FMINV — half-precision</a>	FEAT_FP16
0	11	01100	UNALLOCATED	-
0	11	01111	UNALLOCATED	-
1		00011	<a href="#">UADDLV</a>	-
1		01010	<a href="#">UMAXV</a>	-

Decode fields			Instruction Details	Feature
U	size	opcode		
1		11010	<a href="#">UMINV</a>	-
1		11011	UNALLOCATED	-
1	0x	01100	<a href="#">FMAXNMV</a> — <a href="#">single-precision and double-precision</a>	-
1	0x	01111	<a href="#">FMAXV</a> — <a href="#">single-precision and double-precision</a>	-
1	1x	01100	<a href="#">FMINNMV</a> — <a href="#">single-precision and double-precision</a>	-
1	1x	01111	<a href="#">FMINV</a> — <a href="#">single-precision and double-precision</a>	-

**Advanced SIMD three different**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	size	1	Rm						opcode				0	0	Rn						Rd			

Decode fields		Instruction Details
U	opcode	
	1111	UNALLOCATED
0	0000	<a href="#">SADDL, SADDL2</a>
0	0001	<a href="#">SADDW, SADDW2</a>
0	0010	<a href="#">SSUBL, SSUBL2</a>
0	0011	<a href="#">SSUBW, SSUBW2</a>
0	0100	<a href="#">ADDHN, ADDHN2</a>
0	0101	<a href="#">SABAL, SABAL2</a>
0	0110	<a href="#">SUBHN, SUBHN2</a>
0	0111	<a href="#">SABDL, SABDL2</a>
0	1000	<a href="#">SMLAL, SMLAL2 (vector)</a>
0	1001	<a href="#">SQDMLAL, SQDMLAL2 (vector)</a>
0	1010	<a href="#">SMLSL, SMLSL2 (vector)</a>
0	1011	<a href="#">SQDMLSL, SQDMLSL2 (vector)</a>
0	1100	<a href="#">SMULL, SMULL2 (vector)</a>
0	1101	<a href="#">SQDMULL, SQDMULL2 (vector)</a>
0	1110	<a href="#">PMULL, PMULL2</a>
1	0000	<a href="#">UADDL, UADDL2</a>
1	0001	<a href="#">UADDW, UADDW2</a>
1	0010	<a href="#">USUBL, USUBL2</a>
1	0011	<a href="#">USUBW, USUBW2</a>
1	0100	<a href="#">RADDHN, RADDHN2</a>
1	0101	<a href="#">UABAL, UABAL2</a>
1	0110	<a href="#">RSUBHN, RSUBHN2</a>
1	0111	<a href="#">UABDL, UABDL2</a>
1	1000	<a href="#">UMLAL, UMLAL2 (vector)</a>
1	1001	UNALLOCATED
1	1010	<a href="#">UMLSL, UMLSL2 (vector)</a>
1	1011	UNALLOCATED
1	1100	<a href="#">UMULL, UMULL2 (vector)</a>
1	1101	UNALLOCATED
1	1110	UNALLOCATED

## Advanced SIMD three same

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	size	1		Rm								opcode		1										Rd

Decode fields			Instruction Details		Feature
U	size	opcode			
0		00000	<a href="#">SHADD</a>		-
0		00001	<a href="#">SQADD</a>		-
0		00010	<a href="#">SRHADD</a>		-
0		00100	<a href="#">SHSUB</a>		-
0		00101	<a href="#">SQSUB</a>		-
0		00110	<a href="#">CMGT (register)</a>		-
0		00111	<a href="#">CMGE (register)</a>		-
0		01000	<a href="#">SSHL</a>		-
0		01001	<a href="#">SQSHL (register)</a>		-
0		01010	<a href="#">SRSHL</a>		-
0		01011	<a href="#">SQRSHL</a>		-
0		01100	<a href="#">SMAX</a>		-
0		01101	<a href="#">SMIN</a>		-
0		01110	<a href="#">SABD</a>		-
0		01111	<a href="#">SABA</a>		-
0		10000	<a href="#">ADD (vector)</a>		-
0		10001	<a href="#">CMTST</a>		-
0		10010	<a href="#">MLA (vector)</a>		-
0		10011	<a href="#">MUL (vector)</a>		-
0		10100	<a href="#">SMAXP</a>		-
0		10101	<a href="#">SMINP</a>		-
0		10110	<a href="#">SQDMULH (vector)</a>		-
0		10111	<a href="#">ADDP (vector)</a>		-
0	0x	11000	<a href="#">FMAXNM (vector)</a>		-
0	0x	11001	<a href="#">FMLA (vector)</a>		-
0	0x	11010	<a href="#">FADD (vector)</a>		-
0	0x	11011	<a href="#">FMULX</a>		-
0	0x	11100	<a href="#">FCMEQ (register)</a>		-
0	0x	11110	<a href="#">FMAX (vector)</a>		-
0	0x	11111	<a href="#">FRECPS</a>		-
0	00	00011	<a href="#">AND (vector)</a>		-
0	00	11101	<a href="#">FMLAL, FMLAL2 (vector) — FMLAL</a>		FEAT_FHM
0	01	00011	<a href="#">BIC (vector, register)</a>		-
0	01	11101	UNALLOCATED		-
0	1x	11000	<a href="#">FMINNM (vector)</a>		-
0	1x	11001	<a href="#">FMLS (vector)</a>		-
0	1x	11010	<a href="#">FSUB (vector)</a>		-
0	1x	11011	UNALLOCATED		-
0	1x	11100	UNALLOCATED		-
0	1x	11110	<a href="#">FMIN (vector)</a>		-
0	1x	11111	<a href="#">FRSQRTS</a>		-
0	10	00011	<a href="#">ORR (vector, register)</a>		-

Decode fields			Instruction Details	Feature
U	size	opcode		
0	10	11101	<a href="#">FMLSL, FMLSL2 (vector) — FMLSL</a>	FEAT_FHM
0	11	00011	<a href="#">ORN (vector)</a>	-
0	11	11101	UNALLOCATED	-
1		00000	<a href="#">UHADD</a>	-
1		00001	<a href="#">UQADD</a>	-
1		00010	<a href="#">URHADD</a>	-
1		00100	<a href="#">UHSUB</a>	-
1		00101	<a href="#">UQSUB</a>	-
1		00110	<a href="#">CMHI (register)</a>	-
1		00111	<a href="#">CMHS (register)</a>	-
1		01000	<a href="#">USHL</a>	-
1		01001	<a href="#">UQSHL (register)</a>	-
1		01010	<a href="#">URSHL</a>	-
1		01011	<a href="#">UQRSHL</a>	-
1		01100	<a href="#">UMAX</a>	-
1		01101	<a href="#">UMIN</a>	-
1		01110	<a href="#">UABD</a>	-
1		01111	<a href="#">UABA</a>	-
1		10000	<a href="#">SUB (vector)</a>	-
1		10001	<a href="#">CMEQ (register)</a>	-
1		10010	<a href="#">MLS (vector)</a>	-
1		10011	<a href="#">PMUL</a>	-
1		10100	<a href="#">UMAXP</a>	-
1		10101	<a href="#">UMINP</a>	-
1		10110	<a href="#">SQRDMULH (vector)</a>	-
1		10111	UNALLOCATED	-
1	0x	11000	<a href="#">FMAXNMP (vector)</a>	-
1	0x	11010	<a href="#">FADDP (vector)</a>	-
1	0x	11011	<a href="#">FMUL (vector)</a>	-
1	0x	11100	<a href="#">FCMGE (register)</a>	-
1	0x	11101	<a href="#">FACGE</a>	-
1	0x	11110	<a href="#">FMAXP (vector)</a>	-
1	0x	11111	<a href="#">FDIV (vector)</a>	-
1	00	00011	<a href="#">EOR (vector)</a>	-
1	00	11001	<a href="#">FMLAL, FMLAL2 (vector) — FMLAL2</a>	FEAT_FHM
1	01	00011	<a href="#">BSL</a>	-
1	01	11001	UNALLOCATED	-
1	1x	11000	<a href="#">FMINNMP (vector)</a>	-
1	1x	11010	<a href="#">FABD</a>	-
1	1x	11011	UNALLOCATED	-
1	1x	11100	<a href="#">FCMGT (register)</a>	-
1	1x	11101	<a href="#">FACGT</a>	-
1	1x	11110	<a href="#">FMINP (vector)</a>	-
1	1x	11111	UNALLOCATED	-
1	10	00011	<a href="#">BIT</a>	-
1	10	11001	<a href="#">FMLSL, FMLSL2 (vector) — FMLSL2</a>	FEAT_FHM
1	11	00011	<a href="#">BIF</a>	-

Decode fields			Instruction Details	Feature
U	size	opcode		
1	11	11001	UNALLOCATED	-

### Advanced SIMD modified immediate

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	op	0	1	1	1	1	0	0	0	0	0	a	b	c	cmode			o2	1	d	e	f	g	h	Rd					

Decode fields				Instruction Details	Feature
Q	op	cmode	o2		
	0	0xxx	1	UNALLOCATED	-
	0	0xx0	0	<a href="#">MOVI — 32-bit shifted immediate</a>	-
	0	0xx1	0	<a href="#">ORR (vector, immediate) — 32-bit</a>	-
	0	10xx	1	UNALLOCATED	-
	0	10x0	0	<a href="#">MOVI — 16-bit shifted immediate</a>	-
	0	10x1	0	<a href="#">ORR (vector, immediate) — 16-bit</a>	-
	0	110x	0	<a href="#">MOVI — 32-bit shifting ones</a>	-
	0	110x	1	UNALLOCATED	-
	0	1110	0	<a href="#">MOVI — 8-bit</a>	-
	0	1110	1	UNALLOCATED	-
	0	1111	0	<a href="#">FMOV (vector, immediate) — single-precision</a>	-
	0	1111	1	<a href="#">FMOV (vector, immediate) — half-precision</a>	FEAT_FP16
	1		1	UNALLOCATED	-
	1	0xx0	0	<a href="#">MVNI — 32-bit shifted immediate</a>	-
	1	0xx1	0	<a href="#">BIC (vector, immediate) — 32-bit</a>	-
	1	10x0	0	<a href="#">MVNI — 16-bit shifted immediate</a>	-
	1	10x1	0	<a href="#">BIC (vector, immediate) — 16-bit</a>	-
	1	110x	0	<a href="#">MVNI — 32-bit shifting ones</a>	-
0	1	1110	0	<a href="#">MOVI — 64-bit scalar</a>	-
0	1	1111	0	UNALLOCATED	-
1	1	1110	0	<a href="#">MOVI — 64-bit vector</a>	-
1	1	1111	0	<a href="#">FMOV (vector, immediate) — double-precision</a>	-

### Advanced SIMD shift by immediate

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	1	0	!= 0000			immb			opcode			1	Rn				Rd								
immh																															

The following constraints also apply to this encoding: immh != 0000 && immh != 0000

Decode fields		Instruction Details
U	opcode	
	00001	UNALLOCATED
	00011	UNALLOCATED
	00101	UNALLOCATED
	00111	UNALLOCATED
	01001	UNALLOCATED

Decode fields		Instruction Details
U	opcode	
	01011	UNALLOCATED
	01101	UNALLOCATED
	01111	UNALLOCATED
	10101	UNALLOCATED
	1011x	UNALLOCATED
	110xx	UNALLOCATED
	11101	UNALLOCATED
	11110	UNALLOCATED
0	00000	<a href="#">SSHR</a>
0	00010	<a href="#">SSRA</a>
0	00100	<a href="#">SRSHR</a>
0	00110	<a href="#">SRSRA</a>
0	01000	UNALLOCATED
0	01010	<a href="#">SHL</a>
0	01100	UNALLOCATED
0	01110	<a href="#">SQSHL (immediate)</a>
0	10000	<a href="#">SHRN, SHRN2</a>
0	10001	<a href="#">RSHRN, RSHRN2</a>
0	10010	<a href="#">SQSHRN, SQSHRN2</a>
0	10011	<a href="#">SQRSHRN, SQRSHRN2</a>
0	10100	<a href="#">SSHLL, SSHLL2</a>
0	11100	<a href="#">SCVTF (vector, fixed-point)</a>
0	11111	<a href="#">FCVTZS (vector, fixed-point)</a>
1	00000	<a href="#">USHR</a>
1	00010	<a href="#">USRA</a>
1	00100	<a href="#">URSHR</a>
1	00110	<a href="#">URSRA</a>
1	01000	<a href="#">SRI</a>
1	01010	<a href="#">SLI</a>
1	01100	<a href="#">SQSHLU</a>
1	01110	<a href="#">UQSHL (immediate)</a>
1	10000	<a href="#">SQSHRUN, SQSHRUN2</a>
1	10001	<a href="#">SQRSHRUN, SQRSHRUN2</a>
1	10010	<a href="#">UQSHRN, UQSHRN2</a>
1	10011	<a href="#">UQRSHRN, UQRSHRN2</a>
1	10100	<a href="#">USHLL, USHLL2</a>
1	11100	<a href="#">UCVTF (vector, fixed-point)</a>
1	11111	<a href="#">FCVTZU (vector, fixed-point)</a>

**Advanced SIMD vector x indexed element**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	1	size	L	M	Rm			opcode			H	0	Rn			Rd									

Decode fields			Instruction Details	Feature
U	size	opcode		
	01	1001	UNALLOCATED	-



Decode fields			Instruction Details	Feature
U	size	opcode		
0		0010	<a href="#">SMLAL, SMLAL2 (by element)</a>	-
0		0011	<a href="#">SQDMLAL, SQDMLAL2 (by element)</a>	-
0		0110	<a href="#">SMLSL, SMLSL2 (by element)</a>	-
0		0111	<a href="#">SQDMLSL, SQDMLSL2 (by element)</a>	-
0		1000	<a href="#">MUL (by element)</a>	-
0		1010	<a href="#">SMULL, SMULL2 (by element)</a>	-
0		1011	<a href="#">SQDMULL, SQDMULL2 (by element)</a>	-
0		1100	<a href="#">SQDMULH (by element)</a>	-
0		1101	<a href="#">SQRDMULH (by element)</a>	-
0		1110	<a href="#">SDOT (by element)</a>	FEAT_DotProd
0	0x	0000	UNALLOCATED	-
0	0x	0100	UNALLOCATED	-
0	00	0001	<a href="#">FMLA (by element) — half-precision</a>	FEAT_FP16
0	00	0101	<a href="#">FMLS (by element) — half-precision</a>	FEAT_FP16
0	00	1001	<a href="#">FMUL (by element) — half-precision</a>	FEAT_FP16
0	00	1111	<a href="#">SUDOT (by element)</a>	FEAT_I8MM
0	01	0001	UNALLOCATED	-
0	01	0101	UNALLOCATED	-
0	01	1111	<a href="#">BFDOT (by element)</a>	FEAT_BF16
0	1x	0001	<a href="#">FMLA (by element) — single-precision and double-precision</a>	-
0	1x	0101	<a href="#">FMLS (by element) — single-precision and double-precision</a>	-
0	1x	1001	<a href="#">FMUL (by element) — single-precision and double-precision</a>	-
0	10	0000	<a href="#">FMLAL, FMLAL2 (by element) — FMLAL</a>	FEAT_FHM
0	10	0100	<a href="#">FMLS, FMLS2 (by element) — FMLS</a>	FEAT_FHM
0	10	1111	<a href="#">USDOT (by element)</a>	FEAT_I8MM
0	11	0000	UNALLOCATED	-
0	11	0100	UNALLOCATED	-
0	11	1111	<a href="#">BFMLALB, BFMLALT (by element)</a>	FEAT_BF16
1		0000	<a href="#">MLA (by element)</a>	-
1		0010	<a href="#">UMLAL, UMLAL2 (by element)</a>	-
1		0100	<a href="#">MLS (by element)</a>	-
1		0110	<a href="#">UMLSL, UMLSL2 (by element)</a>	-
1		1010	<a href="#">UMULL, UMULL2 (by element)</a>	-
1		1011	UNALLOCATED	-
1		1101	<a href="#">SQRDMLAH (by element)</a>	FEAT_RDM
1		1110	<a href="#">UDOT (by element)</a>	FEAT_DotProd
1		1111	<a href="#">SQRDMLSH (by element)</a>	FEAT_RDM
1	0x	1000	UNALLOCATED	-
1	0x	1100	UNALLOCATED	-
1	00	0001	UNALLOCATED	-
1	00	0011	UNALLOCATED	-
1	00	0101	UNALLOCATED	-
1	00	0111	UNALLOCATED	-
1	00	1001	<a href="#">FMULX (by element) — half-precision</a>	FEAT_FP16
1	01	0xx1	<a href="#">FCMLA (by element)</a>	FEAT_FCMA
1	1x	1001	<a href="#">FMULX (by element) — single-precision and double-precision</a>	-
1	10	0xx1	<a href="#">FCMLA (by element)</a>	FEAT_FCMA

Decode fields			Instruction Details	Feature
U	size	opcode		
1	10	1000	<a href="#">FMLAL, FMLAL2 (by element)</a> — <a href="#">FMLAL2</a>	FEAT_FHM
1	10	1100	<a href="#">FMLS, FMLS2 (by element)</a> — <a href="#">FMLS2</a>	FEAT_FHM
1	11	0001	UNALLOCATED	-
1	11	0011	UNALLOCATED	-
1	11	0101	UNALLOCATED	-
1	11	0111	UNALLOCATED	-
1	11	1000	UNALLOCATED	-
1	11	1100	UNALLOCATED	-

### Cryptographic three-register, imm2

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	0			Rm			1	0	imm2	opcode												

Decode fields opcode	Instruction Details	Feature
00	<a href="#">SM3TT1A</a>	FEAT_SM3
01	<a href="#">SM3TT1B</a>	FEAT_SM3
10	<a href="#">SM3TT2A</a>	FEAT_SM3
11	<a href="#">SM3TT2B</a>	FEAT_SM3

### Cryptographic three-register SHA 512

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1			Rm			1	0	0	0	opcode											

Decode fields O	opcode	Instruction Details	Feature
0	00	<a href="#">SHA512H</a>	FEAT_SHA512
0	01	<a href="#">SHA512H2</a>	FEAT_SHA512
0	10	<a href="#">SHA512SU1</a>	FEAT_SHA512
0	11	<a href="#">RAX1</a>	FEAT_SHA3
1	00	<a href="#">SM3PARTW1</a>	FEAT_SM3
1	01	<a href="#">SM3PARTW2</a>	FEAT_SM3
1	10	<a href="#">SM4EKEY</a>	FEAT_SM4
1	11	UNALLOCATED	-

### Cryptographic four-register

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	Op0				Rm			0			Ra												

Decode fields Op0	Instruction Details	Feature
00	<a href="#">EOR3</a>	FEAT_SHA3
01	<a href="#">BCAX</a>	FEAT_SHA3

Decode fields Op0	Instruction Details	Feature
10	<a href="#">SM3SS1</a>	FEAT_SM3
11	UNALLOCATED	-

### Cryptographic two-register SHA 512

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	1	1	0	0	0	0	0	0	1	0	0	0	opcode	Rn				Rd						

Decode fields opcode	Instruction Details	Feature
00	<a href="#">SHA512SU0</a>	FEAT_SHA512
01	<a href="#">SM4E</a>	FEAT_SM4
1x	UNALLOCATED	-

### Conversion between floating-point and fixed-point

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	S	1	1	1	1	0	ftype	0	rmode	opcode	scale				Rn				Rd											

Decode fields					Instruction Details		Feature
sf	S	ftype	rmode	opcode	scale		
				1xx		UNALLOCATED	-
			x0	00x		UNALLOCATED	-
			x1	01x		UNALLOCATED	-
			0x	00x		UNALLOCATED	-
			1x	01x		UNALLOCATED	-
		10				UNALLOCATED	-
	1					UNALLOCATED	-
0					0xxxxx	UNALLOCATED	-
0	0	00	00	010		<a href="#">SCVTF (scalar, fixed-point) — 32-bit to single-precision</a>	-
0	0	00	00	011		<a href="#">UCVTF (scalar, fixed-point) — 32-bit to single-precision</a>	-
0	0	00	11	000		<a href="#">FCVTZS (scalar, fixed-point) — single-precision to 32-bit</a>	-
0	0	00	11	001		<a href="#">FCVTZU (scalar, fixed-point) — single-precision to 32-bit</a>	-
0	0	01	00	010		<a href="#">SCVTF (scalar, fixed-point) — 32-bit to double-precision</a>	-
0	0	01	00	011		<a href="#">UCVTF (scalar, fixed-point) — 32-bit to double-precision</a>	-
0	0	01	11	000		<a href="#">FCVTZS (scalar, fixed-point) — double-precision to 32-bit</a>	-
0	0	01	11	001		<a href="#">FCVTZU (scalar, fixed-point) — double-precision to 32-bit</a>	-
0	0	11	00	010		<a href="#">SCVTF (scalar, fixed-point) — 32-bit to half-precision</a>	FEAT_FP16
0	0	11	00	011		<a href="#">UCVTF (scalar, fixed-point) — 32-bit to half-precision</a>	FEAT_FP16
0	0	11	11	000		<a href="#">FCVTZS (scalar, fixed-point) — half-precision to 32-bit</a>	FEAT_FP16

sf	S	Decode fields			scale	Instruction Details	Feature
		ftype	rmode	opcode			
0	0	11	11	001		<a href="#">FCVTZU (scalar, fixed-point) — half-precision to 32-bit</a>	FEAT_FP16
1	0	00	00	010		<a href="#">SCVTF (scalar, fixed-point) — 64-bit to single-precision</a>	-
1	0	00	00	011		<a href="#">UCVTF (scalar, fixed-point) — 64-bit to single-precision</a>	-
1	0	00	11	000		<a href="#">FCVTZS (scalar, fixed-point) — single-precision to 64-bit</a>	-
1	0	00	11	001		<a href="#">FCVTZU (scalar, fixed-point) — single-precision to 64-bit</a>	-
1	0	01	00	010		<a href="#">SCVTF (scalar, fixed-point) — 64-bit to double-precision</a>	-
1	0	01	00	011		<a href="#">UCVTF (scalar, fixed-point) — 64-bit to double-precision</a>	-
1	0	01	11	000		<a href="#">FCVTZS (scalar, fixed-point) — double-precision to 64-bit</a>	-
1	0	01	11	001		<a href="#">FCVTZU (scalar, fixed-point) — double-precision to 64-bit</a>	-
1	0	11	00	010		<a href="#">SCVTF (scalar, fixed-point) — 64-bit to half-precision</a>	FEAT_FP16
1	0	11	00	011		<a href="#">UCVTF (scalar, fixed-point) — 64-bit to half-precision</a>	FEAT_FP16
1	0	11	11	000		<a href="#">FCVTZS (scalar, fixed-point) — half-precision to 64-bit</a>	FEAT_FP16
1	0	11	11	001		<a href="#">FCVTZU (scalar, fixed-point) — half-precision to 64-bit</a>	FEAT_FP16

### Conversion between floating-point and integer

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	S	1	1	1	1	0	f	t	y	p	e	1	r	m	o	d	e	0	0	0	0	0	0	Rn			Rd			

sf	S	Decode fields			Instruction Details	Feature
		ftype	rmode	opcode		
			x1	01x	UNALLOCATED	-
			x1	10x	UNALLOCATED	-
			1x	01x	UNALLOCATED	-
			1x	10x	UNALLOCATED	-
	0	10		0xx	UNALLOCATED	-
	0	10		10x	UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00	x1	11x	UNALLOCATED	-
0	0	00	00	000	<a href="#">FCVTNS (scalar) — single-precision to 32-bit</a>	-
0	0	00	00	001	<a href="#">FCVTNU (scalar) — single-precision to 32-bit</a>	-
0	0	00	00	010	<a href="#">SCVTF (scalar, integer) — 32-bit to single-precision</a>	-
0	0	00	00	011	<a href="#">UCVTF (scalar, integer) — 32-bit to single-precision</a>	-
0	0	00	00	100	<a href="#">FCVTAS (scalar) — single-precision to 32-bit</a>	-
0	0	00	00	101	<a href="#">FCVTAU (scalar) — single-precision to 32-bit</a>	-
0	0	00	00	110	<a href="#">FMOV (general) — single-precision to 32-bit</a>	-
0	0	00	00	111	<a href="#">FMOV (general) — 32-bit to single-precision</a>	-
0	0	00	01	000	<a href="#">FCVTPS (scalar) — single-precision to 32-bit</a>	-
0	0	00	01	001	<a href="#">FCVTPU (scalar) — single-precision to 32-bit</a>	-

sf	S	Decode fields		opcode	Instruction Details	Feature
		ftype	rmode			
0	0	00	1x	11x	UNALLOCATED	-
0	0	00	10	000	<a href="#">FCVTMS (scalar) — single-precision to 32-bit</a>	-
0	0	00	10	001	<a href="#">FCVTMU (scalar) — single-precision to 32-bit</a>	-
0	0	00	11	000	<a href="#">FCVTZS (scalar, integer) — single-precision to 32-bit</a>	-
0	0	00	11	001	<a href="#">FCVTZU (scalar, integer) — single-precision to 32-bit</a>	-
0	0	01	0x	11x	UNALLOCATED	-
0	0	01	00	000	<a href="#">FCVTNS (scalar) — double-precision to 32-bit</a>	-
0	0	01	00	001	<a href="#">FCVTNU (scalar) — double-precision to 32-bit</a>	-
0	0	01	00	010	<a href="#">SCVTF (scalar, integer) — 32-bit to double-precision</a>	-
0	0	01	00	011	<a href="#">UCVTF (scalar, integer) — 32-bit to double-precision</a>	-
0	0	01	00	100	<a href="#">FCVTAS (scalar) — double-precision to 32-bit</a>	-
0	0	01	00	101	<a href="#">FCVTAU (scalar) — double-precision to 32-bit</a>	-
0	0	01	01	000	<a href="#">FCVTPS (scalar) — double-precision to 32-bit</a>	-
0	0	01	01	001	<a href="#">FCVTPU (scalar) — double-precision to 32-bit</a>	-
0	0	01	10	000	<a href="#">FCVTMS (scalar) — double-precision to 32-bit</a>	-
0	0	01	10	001	<a href="#">FCVTMU (scalar) — double-precision to 32-bit</a>	-
0	0	01	10	11x	UNALLOCATED	-
0	0	01	11	000	<a href="#">FCVTZS (scalar, integer) — double-precision to 32-bit</a>	-
0	0	01	11	001	<a href="#">FCVTZU (scalar, integer) — double-precision to 32-bit</a>	-
0	0	01	11	110	<a href="#">FJCVTZS</a>	FEAT_JSCVT
0	0	01	11	111	UNALLOCATED	-
0	0	10		11x	UNALLOCATED	-
0	0	11	00	000	<a href="#">FCVTNS (scalar) — half-precision to 32-bit</a>	FEAT_FP16
0	0	11	00	001	<a href="#">FCVTNU (scalar) — half-precision to 32-bit</a>	FEAT_FP16
0	0	11	00	010	<a href="#">SCVTF (scalar, integer) — 32-bit to half-precision</a>	FEAT_FP16
0	0	11	00	011	<a href="#">UCVTF (scalar, integer) — 32-bit to half-precision</a>	FEAT_FP16
0	0	11	00	100	<a href="#">FCVTAS (scalar) — half-precision to 32-bit</a>	FEAT_FP16
0	0	11	00	101	<a href="#">FCVTAU (scalar) — half-precision to 32-bit</a>	FEAT_FP16
0	0	11	00	110	<a href="#">FMOV (general) — half-precision to 32-bit</a>	FEAT_FP16
0	0	11	00	111	<a href="#">FMOV (general) — 32-bit to half-precision</a>	FEAT_FP16
0	0	11	01	000	<a href="#">FCVTPS (scalar) — half-precision to 32-bit</a>	FEAT_FP16
0	0	11	01	001	<a href="#">FCVTPU (scalar) — half-precision to 32-bit</a>	FEAT_FP16
0	0	11	10	000	<a href="#">FCVTMS (scalar) — half-precision to 32-bit</a>	FEAT_FP16
0	0	11	10	001	<a href="#">FCVTMU (scalar) — half-precision to 32-bit</a>	FEAT_FP16
0	0	11	11	000	<a href="#">FCVTZS (scalar, integer) — half-precision to 32-bit</a>	FEAT_FP16
0	0	11	11	001	<a href="#">FCVTZU (scalar, integer) — half-precision to 32-bit</a>	FEAT_FP16
1	0	00		11x	UNALLOCATED	-
1	0	00	00	000	<a href="#">FCVTNS (scalar) — single-precision to 64-bit</a>	-
1	0	00	00	001	<a href="#">FCVTNU (scalar) — single-precision to 64-bit</a>	-
1	0	00	00	010	<a href="#">SCVTF (scalar, integer) — 64-bit to single-precision</a>	-
1	0	00	00	011	<a href="#">UCVTF (scalar, integer) — 64-bit to single-precision</a>	-
1	0	00	00	100	<a href="#">FCVTAS (scalar) — single-precision to 64-bit</a>	-
1	0	00	00	101	<a href="#">FCVTAU (scalar) — single-precision to 64-bit</a>	-
1	0	00	01	000	<a href="#">FCVTPS (scalar) — single-precision to 64-bit</a>	-
1	0	00	01	001	<a href="#">FCVTPU (scalar) — single-precision to 64-bit</a>	-
1	0	00	10	000	<a href="#">FCVTMS (scalar) — single-precision to 64-bit</a>	-
1	0	00	10	001	<a href="#">FCVTMU (scalar) — single-precision to 64-bit</a>	-

sf	S	Decode fields		opcode	Instruction Details	Feature
		ftype	rmode			
1	0	00	11	000	<a href="#">FCVTZS (scalar, integer)</a> — <a href="#">single-precision to 64-bit</a>	-
1	0	00	11	001	<a href="#">FCVTZU (scalar, integer)</a> — <a href="#">single-precision to 64-bit</a>	-
1	0	01	×1	11×	UNALLOCATED	-
1	0	01	00	000	<a href="#">FCVTNS (scalar)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	00	001	<a href="#">FCVTNU (scalar)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	00	010	<a href="#">SCVTF (scalar, integer)</a> — <a href="#">64-bit to double-precision</a>	-
1	0	01	00	011	<a href="#">UCVTF (scalar, integer)</a> — <a href="#">64-bit to double-precision</a>	-
1	0	01	00	100	<a href="#">FCVTAS (scalar)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	00	101	<a href="#">FCVTAU (scalar)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	00	110	<a href="#">FMOV (general)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	00	111	<a href="#">FMOV (general)</a> — <a href="#">64-bit to double-precision</a>	-
1	0	01	01	000	<a href="#">FCVTPS (scalar)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	01	001	<a href="#">FCVTPU (scalar)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	1×	11×	UNALLOCATED	-
1	0	01	10	000	<a href="#">FCVTMS (scalar)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	10	001	<a href="#">FCVTMU (scalar)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	11	000	<a href="#">FCVTZS (scalar, integer)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	01	11	001	<a href="#">FCVTZU (scalar, integer)</a> — <a href="#">double-precision to 64-bit</a>	-
1	0	10	×0	11×	UNALLOCATED	-
1	0	10	01	110	<a href="#">FMOV (general)</a> — <a href="#">top half of 128-bit to 64-bit</a>	-
1	0	10	01	111	<a href="#">FMOV (general)</a> — <a href="#">64-bit to top half of 128-bit</a>	-
1	0	10	1×	11×	UNALLOCATED	-
1	0	11	00	000	<a href="#">FCVTNS (scalar)</a> — <a href="#">half-precision to 64-bit</a>	FEAT_FP16
1	0	11	00	001	<a href="#">FCVTNU (scalar)</a> — <a href="#">half-precision to 64-bit</a>	FEAT_FP16
1	0	11	00	010	<a href="#">SCVTF (scalar, integer)</a> — <a href="#">64-bit to half-precision</a>	FEAT_FP16
1	0	11	00	011	<a href="#">UCVTF (scalar, integer)</a> — <a href="#">64-bit to half-precision</a>	FEAT_FP16
1	0	11	00	100	<a href="#">FCVTAS (scalar)</a> — <a href="#">half-precision to 64-bit</a>	FEAT_FP16
1	0	11	00	101	<a href="#">FCVTAU (scalar)</a> — <a href="#">half-precision to 64-bit</a>	FEAT_FP16
1	0	11	00	110	<a href="#">FMOV (general)</a> — <a href="#">half-precision to 64-bit</a>	FEAT_FP16
1	0	11	00	111	<a href="#">FMOV (general)</a> — <a href="#">64-bit to half-precision</a>	FEAT_FP16
1	0	11	01	000	<a href="#">FCVTPS (scalar)</a> — <a href="#">half-precision to 64-bit</a>	FEAT_FP16
1	0	11	01	001	<a href="#">FCVTPU (scalar)</a> — <a href="#">half-precision to 64-bit</a>	FEAT_FP16
1	0	11	10	000	<a href="#">FCVTMS (scalar)</a> — <a href="#">half-precision to 64-bit</a>	FEAT_FP16
1	0	11	10	001	<a href="#">FCVTMU (scalar)</a> — <a href="#">half-precision to 64-bit</a>	FEAT_FP16
1	0	11	11	000	<a href="#">FCVTZS (scalar, integer)</a> — <a href="#">half-precision to 64-bit</a>	FEAT_FP16
1	0	11	11	001	<a href="#">FCVTZU (scalar, integer)</a> — <a href="#">half-precision to 64-bit</a>	FEAT_FP16

**Floating-point data-processing (1 source)**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	ftype		1	opcode						1	0	0	0	0	Rn				Rd					

M	Decode fields			Instruction Details	Feature
	S	f <sub>type</sub>	opcode		
			1xxxxxx	UNALLOCATED	-
	1			UNALLOCATED	-
0	0	00	000000	FMOV (register) — <a href="#">single-precision</a>	-

Decode fields				Instruction Details	Feature
M	S	f type	opcode		
0	0	00	000001	<a href="#">FABS (scalar) — single-precision</a>	-
0	0	00	000010	<a href="#">FNEG (scalar) — single-precision</a>	-
0	0	00	000011	<a href="#">FSQRT (scalar) — single-precision</a>	-
0	0	00	000100	UNALLOCATED	-
0	0	00	000101	<a href="#">FCVT — single-precision to double-precision</a>	-
0	0	00	000110	UNALLOCATED	-
0	0	00	000111	<a href="#">FCVT — single-precision to half-precision</a>	-
0	0	00	001000	<a href="#">FRINTN (scalar) — single-precision</a>	-
0	0	00	001001	<a href="#">FRINTP (scalar) — single-precision</a>	-
0	0	00	001010	<a href="#">FRINTM (scalar) — single-precision</a>	-
0	0	00	001011	<a href="#">FRINTZ (scalar) — single-precision</a>	-
0	0	00	001100	<a href="#">FRINTA (scalar) — single-precision</a>	-
0	0	00	001101	UNALLOCATED	-
0	0	00	001110	<a href="#">FRINTX (scalar) — single-precision</a>	-
0	0	00	001111	<a href="#">FRINTI (scalar) — single-precision</a>	-
0	0	00	010000	<a href="#">FRINT32Z (scalar) — single-precision</a>	FEAT_FRINTTS
0	0	00	010001	<a href="#">FRINT32X (scalar) — single-precision</a>	FEAT_FRINTTS
0	0	00	010010	<a href="#">FRINT64Z (scalar) — single-precision</a>	FEAT_FRINTTS
0	0	00	010011	<a href="#">FRINT64X (scalar) — single-precision</a>	FEAT_FRINTTS
0	0	00	0101xx	UNALLOCATED	-
0	0	00	011xxx	UNALLOCATED	-
0	0	01	000000	<a href="#">FMOV (register) — double-precision</a>	-
0	0	01	000001	<a href="#">FABS (scalar) — double-precision</a>	-
0	0	01	000010	<a href="#">FNEG (scalar) — double-precision</a>	-
0	0	01	000011	<a href="#">FSQRT (scalar) — double-precision</a>	-
0	0	01	000100	<a href="#">FCVT — double-precision to single-precision</a>	-
0	0	01	000101	UNALLOCATED	-
0	0	01	000110	<a href="#">BFCVT</a>	FEAT_BF16
0	0	01	000111	<a href="#">FCVT — double-precision to half-precision</a>	-
0	0	01	001000	<a href="#">FRINTN (scalar) — double-precision</a>	-
0	0	01	001001	<a href="#">FRINTP (scalar) — double-precision</a>	-
0	0	01	001010	<a href="#">FRINTM (scalar) — double-precision</a>	-
0	0	01	001011	<a href="#">FRINTZ (scalar) — double-precision</a>	-
0	0	01	001100	<a href="#">FRINTA (scalar) — double-precision</a>	-
0	0	01	001101	UNALLOCATED	-
0	0	01	001110	<a href="#">FRINTX (scalar) — double-precision</a>	-
0	0	01	001111	<a href="#">FRINTI (scalar) — double-precision</a>	-
0	0	01	010000	<a href="#">FRINT32Z (scalar) — double-precision</a>	FEAT_FRINTTS
0	0	01	010001	<a href="#">FRINT32X (scalar) — double-precision</a>	FEAT_FRINTTS
0	0	01	010010	<a href="#">FRINT64Z (scalar) — double-precision</a>	FEAT_FRINTTS
0	0	01	010011	<a href="#">FRINT64X (scalar) — double-precision</a>	FEAT_FRINTTS
0	0	01	0101xx	UNALLOCATED	-
0	0	01	011xxx	UNALLOCATED	-
0	0	10	0xxxxxx	UNALLOCATED	-
0	0	11	000000	<a href="#">FMOV (register) — half-precision</a>	FEAT_FP16
0	0	11	000001	<a href="#">FABS (scalar) — half-precision</a>	FEAT_FP16
0	0	11	000010	<a href="#">FNEG (scalar) — half-precision</a>	FEAT_FP16

Decode fields				Instruction Details	Feature
M	S	fctype	opcode		
0	0	11	000011	<a href="#">FSQRT (scalar) — half-precision</a>	FEAT_FP16
0	0	11	000100	<a href="#">FCVT — half-precision to single-precision</a>	-
0	0	11	000101	<a href="#">FCVT — half-precision to double-precision</a>	-
0	0	11	00011x	UNALLOCATED	-
0	0	11	001000	<a href="#">FRINTN (scalar) — half-precision</a>	FEAT_FP16
0	0	11	001001	<a href="#">FRINTP (scalar) — half-precision</a>	FEAT_FP16
0	0	11	001010	<a href="#">FRINTM (scalar) — half-precision</a>	FEAT_FP16
0	0	11	001011	<a href="#">FRINTZ (scalar) — half-precision</a>	FEAT_FP16
0	0	11	001100	<a href="#">FRINTA (scalar) — half-precision</a>	FEAT_FP16
0	0	11	001101	UNALLOCATED	-
0	0	11	001110	<a href="#">FRINTX (scalar) — half-precision</a>	FEAT_FP16
0	0	11	001111	<a href="#">FRINTI (scalar) — half-precision</a>	FEAT_FP16
0	0	11	01xxxx	UNALLOCATED	-
1				UNALLOCATED	-

### Floating-point compare

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	fctype	1						Rm		op		1	0	0	0				Rn					opcode2

Decode fields					Instruction Details	Feature
M	S	fctype	op	opcode2		
				xxxx1	UNALLOCATED	-
				xxx1x	UNALLOCATED	-
				xx1xx	UNALLOCATED	-
			x1		UNALLOCATED	-
			1x		UNALLOCATED	-
		10			UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00	00	00000	<a href="#">FCMP</a>	-
0	0	00	00	01000	<a href="#">FCMP</a>	-
0	0	00	00	10000	<a href="#">FCMPE</a>	-
0	0	00	00	11000	<a href="#">FCMPE</a>	-
0	0	01	00	00000	<a href="#">FCMP</a>	-
0	0	01	00	01000	<a href="#">FCMP</a>	-
0	0	01	00	10000	<a href="#">FCMPE</a>	-
0	0	01	00	11000	<a href="#">FCMPE</a>	-
0	0	11	00	00000	<a href="#">FCMP</a>	FEAT_FP16
0	0	11	00	01000	<a href="#">FCMP</a>	FEAT_FP16
0	0	11	00	10000	<a href="#">FCMPE</a>	FEAT_FP16
0	0	11	00	11000	<a href="#">FCMPE</a>	FEAT_FP16
1					UNALLOCATED	-

### Floating-point immediate

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	f	t	y	p	e	1	imm8					1	0	0	imm5					Rd				

Decode fields				Instruction Details		Feature
M	S	f	t	y	p	e
				xxxx1	UNALLOCATED	-
				xxx1x	UNALLOCATED	-
				xx1xx	UNALLOCATED	-
				x1xxx	UNALLOCATED	-
				1xxxx	UNALLOCATED	-
		10			UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00		00000	<a href="#">FMOV (scalar, immediate) — single-precision</a>	-
0	0	01		00000	<a href="#">FMOV (scalar, immediate) — double-precision</a>	-
0	0	11		00000	<a href="#">FMOV (scalar, immediate) — half-precision</a>	FEAT_FP16
1					UNALLOCATED	-

### Floating-point conditional compare

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	f	t	y	p	e	1	Rm			cond			0	1	Rn			op	nzc			v		

Decode fields				Instruction Details		Feature
M	S	f	t	y	p	e
		10			UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00	0		<a href="#">FCCMP — single-precision</a>	-
0	0	00	1		<a href="#">FCCMPE — single-precision</a>	-
0	0	01	0		<a href="#">FCCMP — double-precision</a>	-
0	0	01	1		<a href="#">FCCMPE — double-precision</a>	-
0	0	11	0		<a href="#">FCCMP — half-precision</a>	FEAT_FP16
0	0	11	1		<a href="#">FCCMPE — half-precision</a>	FEAT_FP16
1					UNALLOCATED	-

### Floating-point data-processing (2 source)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	ftype	1	Rm			opcode			1	0	Rn			Rd										

Decode fields				Instruction Details		Feature
M	S	f	t	y	p	e
				1xx1	UNALLOCATED	-
				1x1x	UNALLOCATED	-
				11xx	UNALLOCATED	-
		10			UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00		0000	<a href="#">FMUL (scalar) — single-precision</a>	-
0	0	00		0001	<a href="#">FDIV (scalar) — single-precision</a>	-
0	0	00		0010	<a href="#">FADD (scalar) — single-precision</a>	-
0	0	00		0011	<a href="#">FSUB (scalar) — single-precision</a>	-

Decode fields				Instruction Details	Feature
M	S	fctype	opcode		
0	0	00	0100	<a href="#">FMAX (scalar) — single-precision</a>	-
0	0	00	0101	<a href="#">FMIN (scalar) — single-precision</a>	-
0	0	00	0110	<a href="#">FMAXNM (scalar) — single-precision</a>	-
0	0	00	0111	<a href="#">FMINNM (scalar) — single-precision</a>	-
0	0	00	1000	<a href="#">FNMUL (scalar) — single-precision</a>	-
0	0	01	0000	<a href="#">FMUL (scalar) — double-precision</a>	-
0	0	01	0001	<a href="#">FDIV (scalar) — double-precision</a>	-
0	0	01	0010	<a href="#">FADD (scalar) — double-precision</a>	-
0	0	01	0011	<a href="#">FSUB (scalar) — double-precision</a>	-
0	0	01	0100	<a href="#">FMAX (scalar) — double-precision</a>	-
0	0	01	0101	<a href="#">FMIN (scalar) — double-precision</a>	-
0	0	01	0110	<a href="#">FMAXNM (scalar) — double-precision</a>	-
0	0	01	0111	<a href="#">FMINNM (scalar) — double-precision</a>	-
0	0	01	1000	<a href="#">FNMUL (scalar) — double-precision</a>	-
0	0	11	0000	<a href="#">FMUL (scalar) — half-precision</a>	FEAT_FP16
0	0	11	0001	<a href="#">FDIV (scalar) — half-precision</a>	FEAT_FP16
0	0	11	0010	<a href="#">FADD (scalar) — half-precision</a>	FEAT_FP16
0	0	11	0011	<a href="#">FSUB (scalar) — half-precision</a>	FEAT_FP16
0	0	11	0100	<a href="#">FMAX (scalar) — half-precision</a>	FEAT_FP16
0	0	11	0101	<a href="#">FMIN (scalar) — half-precision</a>	FEAT_FP16
0	0	11	0110	<a href="#">FMAXNM (scalar) — half-precision</a>	FEAT_FP16
0	0	11	0111	<a href="#">FMINNM (scalar) — half-precision</a>	FEAT_FP16
0	0	11	1000	<a href="#">FNMUL (scalar) — half-precision</a>	FEAT_FP16
1				UNALLOCATED	-

### Floating-point conditional select

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	fctype	1				Rm						cond		1	1									Rd

Decode fields			Instruction Details	Feature
M	S	fctype		
		10	UNALLOCATED	-
	1		UNALLOCATED	-
0	0	00	<a href="#">FCSEL — single-precision</a>	-
0	0	01	<a href="#">FCSEL — double-precision</a>	-
0	0	11	<a href="#">FCSEL — half-precision</a>	FEAT_FP16
1			UNALLOCATED	-

### Floating-point data-processing (3 source)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	1	fctype	o1				Rm			o0															Rd

M	Decode fields			Instruction Details		Feature
	S	f <sub>type</sub>	o <sub>1</sub>	o <sub>0</sub>		
		10			UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00	0	0	<a href="#">FMADD</a> — <a href="#">single-precision</a>	-
0	0	00	0	1	<a href="#">FMSUB</a> — <a href="#">single-precision</a>	-
0	0	00	1	0	<a href="#">FNMADD</a> — <a href="#">single-precision</a>	-
0	0	00	1	1	<a href="#">FNMSUB</a> — <a href="#">single-precision</a>	-
0	0	01	0	0	<a href="#">FMADD</a> — <a href="#">double-precision</a>	-
0	0	01	0	1	<a href="#">FMSUB</a> — <a href="#">double-precision</a>	-
0	0	01	1	0	<a href="#">FNMADD</a> — <a href="#">double-precision</a>	-
0	0	01	1	1	<a href="#">FNMSUB</a> — <a href="#">double-precision</a>	-
0	0	11	0	0	<a href="#">FMADD</a> — <a href="#">half-precision</a>	FEAT_FP16
0	0	11	0	1	<a href="#">FMSUB</a> — <a href="#">half-precision</a>	FEAT_FP16
0	0	11	1	0	<a href="#">FNMADD</a> — <a href="#">half-precision</a>	FEAT_FP16
0	0	11	1	1	<a href="#">FNMSUB</a> — <a href="#">half-precision</a>	FEAT_FP16
1					UNALLOCATED	-

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# Shared Pseudocode Functions

This page displays common pseudocode functions shared by many pages

## Pseudocodes

### Library pseudocode for aarch32/at/AArch32.AT

```
// AArch32.AT()
// =====
// Perform address translation as per AT instructions.

AArch32.AT(bits(32) vaddress, TranslationStage stage_in, bits(2) el, ATAccess ataccess)
    TranslationStage stage = stage_in;
    SecurityState ss;
    Regime regime;
    boolean eae;

    // ATs1Hx instructions
    if el == EL2 then
        regime = Regime\_EL2;
        eae = TRUE;
        ss = SS\_NonSecure;

    // ATs1Cxx instructions
    elseif stage == TranslationStage\_1 || (stage == TranslationStage\_12 && !HaveEL(EL2)) then
        stage = TranslationStage\_1;
        ss = SecurityStateAtEL(PSTATE.EL);
        regime = if ss == SS\_Secure && ELUsingAArch32(EL3) then Regime\_EL30 else Regime\_EL10;
        eae = TTBCR.EAE == '1';

    // ATs12NSOxx instructions
    else
        regime = Regime\_EL10;
        eae = if HaveAArch32EL(EL3) then TTBCR_NS.EAE == '1' else TTBCR.EAE == '1';
        ss = SS\_NonSecure;

    AddressDescriptor addrdesc;
    SDFTYPE sdftype;
    boolean aligned = TRUE;
    bit supersection = '0';

    accdesc = CreateAccDescAT(ss, el, ataccess);

    // Prepare fault fields in case a fault is detected
    fault = NoFault(accdesc);

    if eae then
        (fault, addrdesc) = AArch32.S1TranslateLD(fault, regime, vaddress, aligned, accdesc);
    else
        (fault, addrdesc, sdftype) = AArch32.S1TranslateSD(fault, regime, vaddress, aligned,
            accdesc);
        supersection = if sdftype == SDFTYPE\_Supersection then '1' else '0';

    // ATs12NSOxx instructions
    if stage == TranslationStage\_12 && fault.statuscode == Fault\_None then
        (fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, aligned, accdesc);

    if fault.statuscode != Fault\_None then
        // Take exception on External abort or when a fault occurs on translation table walk
        if IsExternalAbort(fault) || (PSTATE.EL == EL1 && EL2Enabled() && fault.s2fslwalk) then
            PAR = bits(64) UNKNOWN;
            AArch32.Abort(vaddress, fault);

    addrdesc.fault = fault;

    if (eae || (stage == TranslationStage\_12 && (HCR.VM == '1' || HCR.DC == '1'))
        || (stage == TranslationStage\_1 && el != EL2 && PSTATE.EL == EL2)) then
        AArch32.EncodePARLD(addrdesc, ss);
    else
        AArch32.EncodePARSD(addrdesc, supersection, ss);
    return;
```

## Library pseudocode for aarch32/at/AArch32.EncodePARLD

```
// AArch32.EncodePARLD()
// =====
// Returns 64-bit format PAR on address translation instruction.

AArch32.EncodePARLD(AddressDescriptor addrdesc, SecurityState ss)

    if !IsFault(addrdesc) then
        bit ns;
        if ss == SS_NonSecure then
            ns = bit UNKNOWN;
        elsif addrdesc.paddress.paspace == PAS_Secure then
            ns = '0';
        else
            ns = '1';
        PAR.F = '0';
        PAR.SH = ReportedPARShareability(PAREncodeShareability(addrdesc.memattrs));
        PAR.NS = ns;
        PAR<10> = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR"; // IMPDEF
        PAR.LPAE = '1';
        PAR.PA = addrdesc.paddress.address<39:12>;
        PAR.ATTR = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
    else
        PAR.F = '1';
        PAR.FST = AArch32.PARFaultStatusLD(addrdesc.fault);
        PAR.S2WLK = if addrdesc.fault.s2fslwalk then '1' else '0';
        PAR.FSTAGE = if addrdesc.fault.secondstage then '1' else '0';
        PAR.LPAE = '1';
        PAR<63:48> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR"; // IMPDEF
    return;
```

## Library pseudocode for aarch32/at/AArch32.EncodePARSD

```
// AArch32.EncodePARSD()
// =====
// Returns 32-bit format PAR on address translation instruction.

AArch32.EncodePARSD(AddressDescriptor addrdesc_in, bit supersection, SecurityState ss)
    AddressDescriptor addrdesc = addrdesc_in;
    if !IsFault(addrdesc) then
        if (addrdesc.memattrs.memtype == MemType_Device ||
            (addrdesc.memattrs.inner.attrs == MemAttr_NC &&
             addrdesc.memattrs.outer.attrs == MemAttr_NC)) then
            addrdesc.memattrs.shareability = Shareability_OSH;
        bit ns;
        if ss == SS_NonSecure then
            ns = bit UNKNOWN;
        elsif addrdesc.paddress.paspace == PAS_Secure then
            ns = '0';
        else
            ns = '1';
        bits(2) sh = if addrdesc.memattrs.shareability != Shareability_NSH then '01' else '00';
        PAR.F = '0';
        PAR.SS = supersection;
        PAR.Outer = AArch32.ReportedOuterAttrs(AArch32.PAROuterAttrs(addrdesc.memattrs));
        PAR.Inner = AArch32.ReportedInnerAttrs(AArch32.PARInnerAttrs(addrdesc.memattrs));
        PAR.SH = ReportedPARShareability(sh);
        PAR<8> = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR"; // IMPDEF
        PAR.NS = ns;
        PAR.NOS = if addrdesc.memattrs.shareability == Shareability_OSH then '0' else '1';
        PAR.LPAE = '0';
        PAR.PA = addrdesc.paddress.address<39:12>;
    else
        PAR.F = '1';
        PAR.FST = AArch32.PARFaultStatusSD(addrdesc.fault);
        PAR.LPAE = '0';
        PAR<31:16> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR"; // IMPDEF
    return;
```

## Library pseudocode for aarch32/at/AArch32.PARFaultStatusLD

```
// AArch32.PARFaultStatusLD()
// =====
// Fault status field decoding of 64-bit PAR

bits(6) AArch32.PARFaultStatusLD(FaultRecord fault)
    bits(6) syndrome;

    if fault.statuscode == Fault\_Domain then
        // Report Domain fault
        assert fault.level IN {1,2};
        syndrome<1:0> = if fault.level == 1 then '01' else '10';
        syndrome<5:2> = '1111';
    else
        syndrome = EncodeLDFSC(fault.statuscode, fault.level);
    return syndrome;
```

## Library pseudocode for aarch32/at/AArch32.PARFaultStatusSD

```
// AArch32.PARFaultStatusSD()
// =====
// Fault status field decoding of 32-bit PAR.

bits(6) AArch32.PARFaultStatusSD(FaultRecord fault)
    bits(6) syndrome;

    syndrome<5> = if IsExternalAbort(fault) then fault.extflag else '0';
    syndrome<4:0> = EncodeSDFSC(fault.statuscode, fault.level);
    return syndrome;
```

## Library pseudocode for aarch32/at/AArch32.PARInnerAttrs

```
// AArch32.PARInnerAttrs()
// =====
// Convert orthogonal attributes and hints to 32-bit PAR Inner field.

bits(3) AArch32.PARInnerAttrs(MemoryAttributes memattrs)
    bits(3) result;

    if memattrs.memtype == MemType\_Device then
        if memattrs.device == DeviceType\_nGnRnE then
            result = '001'; // Non-cacheable
        elsif memattrs.device == DeviceType\_nGnRE then
            result = '011'; // Non-cacheable
    else
        MemAttrHints inner = memattrs.inner;
        if inner.attrs == MemAttr\_NC then
            result = '000'; // Non-cacheable
        elsif inner.attrs == MemAttr\_WB && inner.hints<0> == '1' then
            result = '101'; // Write-Back, Write-Allocate
        elsif inner.attrs == MemAttr\_WT then
            result = '110'; // Write-Through
        elsif inner.attrs == MemAttr\_WB && inner.hints<0> == '0' then
            result = '111'; // Write-Back, no Write-Allocate
    return result;
```

### Library pseudocode for aarch32/at/AArch32.PAROuterAttrs

```
// AArch32.PAROuterAttrs()
// =====
// Convert orthogonal attributes and hints to 32-bit PAR Outer field.

bits(2) AArch32.PAROuterAttrs(MemoryAttributes memattrs)
    bits(2) result;

    if memattrs.memtype == MemType_Device then
        result = bits(2) UNKNOWN;
    else
        MemAttrHints outer = memattrs.outer;
        if outer.attrs == MemAttr_NC then
            result = '00'; // Non-cacheable
        elseif outer.attrs == MemAttr_WB && outer.hints<0> == '1' then
            result = '01'; // Write-Back, Write-Allocate
        elseif outer.attrs == MemAttr_WT && outer.hints<0> == '0' then
            result = '10'; // Write-Through, no Write-Allocate
        elseif outer.attrs == MemAttr_WB && outer.hints<0> == '0' then
            result = '11'; // Write-Back, no Write-Allocate
        return result;
```

### Library pseudocode for aarch32/at/AArch32.ReportedInnerAttrs

```
// AArch32.ReportedInnerAttrs()
// =====
// The value returned in this field can be the resulting attribute, as determined by any permitted
// implementation choices and any applicable configuration bits, instead of the value that appears
// in the translation table descriptor.

bits(3) AArch32.ReportedInnerAttrs(bits(3) attrs);
```

### Library pseudocode for aarch32/at/AArch32.ReportedOuterAttrs

```
// AArch32.ReportedOuterAttrs()
// =====
// The value returned in this field can be the resulting attribute, as determined by any permitted
// implementation choices and any applicable configuration bits, instead of the value that appears
// in the translation table descriptor.

bits(2) AArch32.ReportedOuterAttrs(bits(2) attrs);
```





```

// AArch32.DC()
// =====
// Perform Data Cache Operation.

AArch32.DC(bits(32) regval, CacheOp cacheop, CacheOpScope opscope)
    CacheRecord cache;

cache.acctype = AccessType\_DC;
cache.cacheop = cacheop;
cache.opscope = opscope;
cache.cachetype = CacheType\_Data;
cache.security = SecurityStateAtEL(PSTATE.EL);

if opscope == CacheOpScope\_SetWay then
    cache.shareability = Shareability\_NSH;
    (cache.setnum, cache.waynum, cache.level) = DecodeSW(ZeroExtend(regval, 64),
        CacheType\_Data);

    if (cacheop == CacheOp\_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
        ((ELUsingAArch32(EL2) && (HCR_EL2.SWIO == '1' || HCR_EL2.<DC,VM> != '00')) ||
        (ELUsingAArch32(EL2) && (HCR.SWIO == '1' || HCR.<DC,VM> != '00')))) then
        cache.cacheop = CacheOp\_CleanInvalidate;
        CACHE\_OP(cache);
        return;

if EL2Enabled() then
    if PSTATE.EL IN {EL0, EL1} then
        cache.is_vmid_valid = TRUE;
        cache.vmid = VMID[];
    else
        cache.is_vmid_valid = FALSE;
else
    cache.is_vmid_valid = FALSE;

if PSTATE.EL == EL0 then
    cache.is_asid_valid = TRUE;
    cache.asid = ASID[];
else
    cache.is_asid_valid = FALSE;

need_translate = DCInstNeedsTranslation(opscope);
vaddress = regval;

size = 0; // by default no watchpoint address
if cacheop == CacheOp\_Invalidate then
    size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
    assert size >= 4*(2^(UInt(CTR_EL0.DminLine))) && size <= 2048;
    assert UInt(size<32:0> AND (size-1)<32:0>) == 0; // size is power of 2
    vaddress = Align(regval, size);

cache.translated = need_translate;
cache.vaddress = ZeroExtend(vaddress, 64);

if need_translate then
    boolean aligned = TRUE;
    AccessDescriptor accdesc = CreateAccDescDC(cache);
    AddressDescriptor memaddrdesc = AArch32.TranslateAddress(vaddress, accdesc, aligned, size);
    if IsFault(memaddrdesc) then
        AArch32.Abort(regval, memaddrdesc.fault);

    cache.paddress = memaddrdesc.paddress;
    if opscope == CacheOpScope\_PoC then
        cache.shareability = memaddrdesc.memattrs.shareability;
    else
        cache.shareability = Shareability\_NSH;
else
    cache.shareability = Shareability\_UNKNOWN;
    cache.paddress = FullAddress\_UNKNOWN;

if (cacheop == CacheOp\_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&

```

```

        ((!ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00') ||
         (ELUsingAArch32(EL2) && HCR.<DC,VM> != '00')) then
    cache.cacheop = CacheOp\_CleanInvalidate;

    CACHE\_OP(cache);
    return;

```

## Library pseudocode for aarch32/debug/VCRMatch/AArch32.VCRMatch

```

// AArch32.VCRMatch()
// =====

boolean AArch32.VCRMatch(bits(32) vaddress)

    boolean match;
    if UsingAArch32() && ELUsingAArch32(EL1) && PSTATE.EL != EL2 then
        // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
        match_word = Zeros(32);

        ss = CurrentSecurityState();
        if vaddress<31:5> == ExcVectorBase()<31:5> then
            if HaveEL(EL3) && ss == SS\_NonSecure then
                match_word<UInt(vaddress<4:2>) + 24> = '1';           // Non-secure vectors
            else
                match_word<UInt(vaddress<4:2>) + 0> = '1';           // Secure vectors (or no EL3)

        if (HaveEL(EL3) && ELUsingAArch32(EL3) && vaddress<31:5> == MVBAR<31:5> &&
            ss == SS\_Secure) then
            match_word<UInt(vaddress<4:2>) + 8> = '1';           // Monitor vectors

        // Mask out bits not corresponding to vectors.
        bits(32) mask;
        if !HaveEL(EL3) then
            mask = '00000000':'00000000':'00000000':'11011110'; // DBGVCR[31:8] are RES0
        elseif !ELUsingAArch32(EL3) then
            mask = '11011110':'00000000':'00000000':'11011110'; // DBGVCR[15:8] are RES0
        else
            mask = '11011110':'00000000':'11011100':'11011110';

        match_word = match_word AND DBGVCR AND mask;
        match = !IsZero(match_word);

        // Check for UNPREDICTABLE case - match on Prefetch Abort and Data Abort vectors
        if !IsZero(match_word<28:27,12:11,4:3>) && DebugTarget() == PSTATE.EL then
            match = ConstrainUnpredictableBool(Unpredictable\_VCMATCHDAPA);

        if !IsZero(vaddress<1:0>) && match then
            match = ConstrainUnpredictableBool(Unpredictable\_VCMATCHHALF);
    else
        match = FALSE;

    return match;

```

## Library pseudocode for aarch32/debug/authentication/AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled

```

// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// =====

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
    // the state of the (DBGEN AND SPIDEN) signal.
    if !HaveEL(EL3) && NonSecureOnlyImplementation() then return FALSE;
    return DBGEN == Signal\_High && SPIDEN == Signal\_High;

```

## Library pseudocode for aarch32/debug/breakpoint/AArch32.BreakpointMatch

```
// AArch32.BreakpointMatch()
// =====
// Breakpoint matching in an AArch32 translation regime.

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress, AccessDescriptor accdesc,
                                           integer size)
    assert ELUsingAArch32\(S1TranslationRegime\)();
    assert n < NumBreakpointsImplemented();

    enabled      = DBGBCR[n].E == '1';
    isbreakpnt   = TRUE;
    linked       = DBGBCR[n].BT IN {'0x01'};
    linked_to    = FALSE;
    linked_n     = UInt(DBGBCR[n].LBN);

    state_match  = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC,
                                     linked, linked_n, isbreakpnt, accdesc);
    (value_match, value_mismatch) = AArch32.BreakpointValueMatch(n, vaddress, linked_to);

    if size == 4 then // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        (match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress + 2, linked_to);

        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable\_BPMATCHHALF);

        if value_mismatch && !mismatch_i then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable\_BPMISMATCHHALF);

    if vaddress<1> == '1' && DBGBCR[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBCR[n]+2.
        if value_match then
            value_match = ConstrainUnpredictableBool(Unpredictable\_BPMATCHHALF);

        if !value_mismatch then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable\_BPMISMATCHHALF);

    match      = value_match && state_match && enabled;
    mismatch   = value_mismatch && state_match && enabled;

    return (match, mismatch);
```



```

// AArch32.BreakpointValueMatch()
// =====
// The first result is whether an Address Match or Context breakpoint is programmed on the
// instruction at "address". The second result is whether an Address Mismatch breakpoint is
// programmed on the instruction, that is, whether the instruction should be stepped.

(boolean, boolean) AArch32.BreakpointValueMatch(integer n_in, bits(32) vaddress, boolean linked_to)

// "n" is the identity of the breakpoint unit to match against.
// "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
// matching breakpoints.
// "linked_to" is TRUE if this is a call from StateMatch for linking.
integer n = n_in;
Constraint c;

// If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
// no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
if n >= NumBreakpointsImplemented() then
    (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1,
                                           Unpredictable_BPNOTIMPL);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return (FALSE, FALSE);

// If this breakpoint is not enabled, it cannot generate a match.
// (This could also happen on a call from StateMatch for linking).
if DBGBCR[n].E == '0' then return (FALSE, FALSE);

dbgtype = DBGBCR[n].BT;

(c, dbgtype) = AArch32.ReservedBreakpointType(n, dbgtype);
if c == Constraint_DISABLED then return (FALSE, FALSE);
// Otherwise the dbgtype value returned by AArch32.ReservedBreakpointType is valid.

// Determine what to compare against.
match_addr      = (dbgtype IN {'0x0x'});
mismatch        = (dbgtype IN {'010x'});
match_vmid      = (dbgtype IN {'10xx'});
match_cid1      = (dbgtype IN {'xx1x'});
match_cid2      = (dbgtype IN {'11xx'});
linking_enabled = (dbgtype IN {'xxx1'});

// If called from StateMatch, is is CONSTRAINED UNPREDICTABLE if the
// breakpoint is not programmed with linking enabled.
if linked_to && !linking_enabled then
    if !ConstrainUnpredictableBool(Unpredictable_BPLINKINGDISABLED) then
        return (FALSE, FALSE);

// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked_to && linking_enabled && !match_addr then
    return (FALSE, FALSE);

boolean bvr_match = FALSE;
boolean bxvr_match = FALSE;

// Do the comparison.
if match_addr then
    integer byte = UInt(vaddress<1:0>);
    assert byte IN {0,2}; // "vaddress" is halfword aligned

    boolean byte_select_match = (DBGBCR[n].BAS<byte> == '1');
    integer top = 31;
    bvr_match = (vaddress<top:2> == DBGBVR[n]<top:2>) && byte_select_match;

elseif match_cid1 then
    bvr_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBVR[n]<31:0>);

if match_vmid then
    bits(16) vmid;
    bits(16) bvr_vmid;

```

```

    if ELUsingAArch32\(EL2\) then
        vmid = ZeroExtend(VTTBR.VMID, 16);
        bvr_vmid = ZeroExtend(DBGDXVR[n]<7:0>, 16);
    elsif !Have16bitVMID() || VTCR_EL2.VS == '0' then
        vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        bvr_vmid = ZeroExtend(DBGDXVR[n]<7:0>, 16);
    else
        vmid = VTTBR_EL2.VMID;
        bvr_vmid = DBGDXVR[n]<15:0>;

    bxvr_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && vmid == bvr_vmid);

    elsif match_cid2 then
        bxvr_match = (PSTATE.EL != EL3 && EL2Enabled() && !ELUsingAArch32\(EL2\) &&
            DBGDXVR[n]<31:0> == CONTEXTIDR_EL2<31:0>);

    bvr_match_valid = (match_addr || match_cid1);
    bxvr_match_valid = (match_vmid || match_cid2);

    match = (!bxvr_match_valid || bxvr_match) && (!bvr_match_valid || bvr_match);

    return (match && !mismatch, !match && mismatch);

```

### Library pseudocode for aarch32/debug/breakpoint/AArch32.ReservedBreakpointType

```

// AArch32.ReservedBreakpointType()
// =====
// Checks if the given DBGBCR<n>.BT value is reserved and will generate Constrained Unpredictable
// behavior, otherwise returns Constraint_NONE.

(Constraint, bits(4)) AArch32.ReservedBreakpointType(integer n, bits(4) bt_in)
    bits(4) bt      = bt_in;
    boolean reserved = FALSE;
    context_aware = n >= (NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented());

    // Address mismatch
    if bt IN {'010x'} && HaltOnBreakpointOrWatchpoint() then
        reserved = TRUE;

    // Context matching
    if !(bt IN {'0x0x'}) && !context_aware then
        reserved = TRUE;

    // EL2 extension
    if bt IN {'1xxx'} && !HaveEL\(EL2\) then
        reserved = TRUE;

    // Context matching
    if bt IN {'011x', '11xx'} && !HaveVirtHostExt() && !HaveV82Debug() then
        reserved = TRUE;

    if reserved then
        Constraint c;
        (c, bt) = ConstrainUnpredictableBits(Unpredictable\_RESBPTYPE, 4);
        assert c IN {Constraint\_DISABLED, Constraint\_UNKNOWN};
        if c == Constraint\_DISABLED then
            return (c, bits(4) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint\_NONE, bt);

```





```

// AArch32.StateMatch()
// =====
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch32.StateMatch(bits(2) ssc_in, bit hmc_in, bits(2) pxc_in, boolean linked_in,
                           integer linked_n_in, boolean isbreakpnt, AccessDescriptor accdesc)

// "ssc_in","hmc_in","pxc_in" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
// "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
// "linked_n_in" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
// "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
// "accdesc" describes the properties of the access being matched.
bit hmc      = hmc_in;
bits(2) ssc   = ssc_in;
bits(2) pxc   = pxc_in;
boolean linked = linked_in;
integer linked_n = linked_n_in;

// If parameters are set to a reserved type, behaves as either disabled or a defined type
Constraint c;
// SSCE value discarded as there is no SSCE bit in AArch32.
(c, ssc, -, hmc, pxc) = CheckValidStateMatch(ssc, '0', hmc, pxc, isbreakpnt);
if c == Constraint\_DISABLED then return FALSE;
// Otherwise the hmc,ssc,pxc values are either valid or the values returned by
// CheckValidStateMatch are valid.

pl2_match = HaveEL(EL2) && ((hmc == '1' && (ssc:pxc != '1000')) || ssc == '11');
pl1_match = pxc<0> == '1';
pl0_match = pxc<1> == '1';
ssu_match = isbreakpnt && hmc == '0' && pxc == '00' && ssc != '11';

boolean priv_match;
if ssu_match then
    priv_match = PSTATE.M IN {M32\_User,M32\_Svc,M32\_System};
else
    case accdesc.el of
        when EL3 priv_match = pl1_match;           // EL3 and EL1 are both PL1
        when EL2 priv_match = pl2_match;
        when EL1 priv_match = pl1_match;
        when EL0 priv_match = pl0_match;

// Security state match
boolean ss_match;
case ssc of
    when '00' ss_match = TRUE;                      // Both
    when '01' ss_match = accdesc.ss == SS\_NonSecure; // Non-secure only
    when '10' ss_match = accdesc.ss == SS\_Secure;    // Secure only
    when '11' ss_match = (hmc == '1' || accdesc.ss == SS\_Secure); // HMC=1 -> Both,
                                                                // HMC=0 -> Secure only

boolean linked_match = FALSE;

if linked then
    // "linked_n" must be an enabled context-aware breakpoint unit.
    // If it is not context-aware then it is CONSTRAINED UNPREDICTABLE whether
    // this gives no match, gives a match without linking, or linked_n is mapped to some
    // UNKNOWN breakpoint that is context-aware.
    if IsContextMatchingBreakpoint(linked_n) then
        (first_ctx_cmp, last_ctx_cmp) = ContextMatchingBreakpointRange();
        (c, linked_n) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp,
                                                       Unpredictable\_BPNOTCTXCMP);
        assert c IN {Constraint\_DISABLED, Constraint\_NONE, Constraint\_UNKNOWN};

        case c of
            when Constraint\_DISABLED return FALSE; // Disabled
            when Constraint\_NONE linked = FALSE;   // No linking
            // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

vaddress = bits(32) UNKNOWN;
linked_to = TRUE;

```

```

(linked_match,-) = AArch32.BreakpointValueMatch(linked_n, vaddress, linked_to);

return priv_match && ss_match && (!linked || linked_match);

```

### Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptions

```

// AArch32.GenerateDebugExceptions()
// =====

boolean AArch32.GenerateDebugExceptions()
    ss = CurrentSecurityState();
    return AArch32.GenerateDebugExceptionsFrom(PSTATE.EL, ss);

```

### Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom

```

// AArch32.GenerateDebugExceptionsFrom()
// =====

boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from_el, SecurityState from_state)

    if !ELUsingAArch32(DebugTargetFrom(from_state)) then
        mask = '0'; // No PSTATE.D in AArch32 state
        return AArch64.GenerateDebugExceptionsFrom(from_el, from_state, mask);

    if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    boolean enabled;
    if HaveEL(EL3) && from_state == SS\_Secure then
        assert from_el != EL2; // Secure EL2 always uses AArch64
        if IsSecureEL2Enabled() then
            // Implies that EL3 and EL2 both using AArch64
            enabled = MDCR_EL3.SDD == '0';
        else
            spd = if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32;
            if spd<1> == '1' then
                enabled = spd<0> == '1';
            else
                // SPD == 0b01 is reserved, but behaves the same as 0b00.
                enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
        if from_el == ELO then enabled = enabled || SDER.SUIDEN == '1';
    else
        enabled = from_el != EL2;

    return enabled;

```

### Library pseudocode for aarch32/debug/pmu/AArch32.ClearEventCounters

```

// AArch32.ClearEventCounters()
// =====
// Zero all the event counters.

AArch32.ClearEventCounters()
    if HaveAArch64() then
        // Force the counter to be cleared as a 64-bit counter.
        AArch64.ClearEventCounters();
        return;

    integer counters = AArch32.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            PMEVCNTR[idx] = Zeros(32);

```

## Library pseudocode for aarch32/debug/pmu/AArch32.GetNumEventCountersAccessible

```
// AArch32.GetNumEventCountersAccessible()
// =====
// Return the number of event counters that can be accessed at the current Exception level.

integer AArch32.GetNumEventCountersAccessible()
    integer n;
    integer total_counters = GetNumEventCounters();
    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled() then
        n = UInt(if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN);
        if n > total_counters || (!HaveFeatHPMN0() && n == 0) then
            (-, n) = ConstrainUnpredictableInteger(0, total_counters,
                Unpredictable\_PMUEVENTCOUNTER);
    else
        n = total_counters;

    return n;
```

## Library pseudocode for aarch32/debug/pmu/AArch32.IncrementCycleCounter

```
// AArch32.IncrementCycleCounter()
// =====
// Increment the cycle counter and possibly set overflow bits.

AArch32.IncrementCycleCounter()
    if !CountPMUEvents(CYCLE\_COUNTER\_ID) then return;
    bit d = PMCR.D; // Check divide-by-64
    bit lc = PMCR.LC;
    // Effective value of 'D' bit is 0 when Effective value of LC is '1'
    if lc == '1' then d = '0';
    if d == '1' && !HasElapsed64Cycles() then return;

    integer old_value = UInt(PMCCNTR);
    integer new_value = old_value + 1;
    PMCCNTR = new_value<63:0>;

    integer ovflw = if lc == '1' then 64 else 32;

    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET.C = '1';
        PMOVSRR.C = '1';

    return;
```

## Library pseudocode for aarch32/debug/pmu/AArch32.IncrementEventCounter

```
// AArch32.IncrementEventCounter()
// =====
// Increment the specified event counter by the specified amount.

AArch32.IncrementEventCounter(integer idx, integer increment)
    if HaveAArch64() then
        // Force the counter to be incremented as a 64-bit counter.
        AArch64.IncrementEventCounter(idx, increment);
        return;

    // In this model, event counters in an AArch32-only implementation are 32 bits and
    // the LP bits are RES0 in this model, even if FEAT_PMUv3p5 is implemented.
    integer old_value;
    integer new_value;
    integer ovflw;

    old_value = UInt(PMEVCNTR[idx]);
    new_value = old_value + PMUCountValue(idx, increment);

    PMEVCNTR[idx] = new_value<31:0>;
    ovflw = 32;

    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET<idx> = '1';
        PMOVSR<idx> = '1';
        // Check for the CHAIN event from an even counter
        if idx<0> == '0' && idx + 1 < GetNumEventCounters() then
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);

    return;
```

## Library pseudocode for aarch32/debug/pmu/AArch32.PMUCycle

```
// AArch32.PMUCycle()
// =====
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

AArch32.PMUCycle()
    if HaveAArch64() then
        AArch64.PMUCycle();
        return;

    if !HavePMUv3() then
        return;

    PMUEvent(PMU_EVENT_CPU_CYCLES);

    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            if CountPMUEvents(idx) then
                integer accumulated = PMUEventAccumulator[idx];
                AArch32.IncrementEventCounter(idx, accumulated);
                PMUEventAccumulator[idx] = 0;
        AArch32.IncrementCycleCounter();
        CheckForPMUOverflow();
```

## Library pseudocode for aarch32/debug/pmu/AArch32.PMUSwIncrement

```
// AArch32.PMUSwIncrement()
// =====
// Generate PMU Events on a write to PMSWINC.

AArch32.PMUSwIncrement(bits(32) sw_incr)
    integer counters = AArch32.GetNumEventCountersAccessible\(\);
    if counters != 0 then
        for idx = 0 to counters - 1
            if sw_incr<idx> == '1' then
                PMUEvent(PMU_EVENT_SW_INCR, 1, idx);
```

## Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState

```
// AArch32.EnterHypModeInDebugState()
// =====
// Take an exception in Debug state to Hyp mode.

AArch32.EnterHypModeInDebugState(ExceptionRecord except)
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState() == SS\_NonSecure && ELUsingAArch32(EL2);

    AArch32.ReportHypEntry(except);
    AArch32.WriteMode(M32\_Hyp);
    SPSR\_curr[] = bits(32) UNKNOWN;
    ELR\_hyp = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    if Havev8p9Debug() then
        DSPSR2 = bits(32) UNKNOWN;
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields();

    EndOfInstruction();
```

## Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterModeInDebugState

```
// AArch32.EnterModeInDebugState()
// =====
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.

AArch32.EnterModeInDebugState(bits(5) target_mode)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR_curr[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    if Havev8p9Debug() then
        DSPSR2 = bits(32) UNKNOWN;
    PSTATE.E = SCTL.R.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTL.R.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSEExt() then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields(); // Update EDSCR processor state flags.

    EndOfInstruction();
```

## Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterMonitorModeInDebugState

```
// AArch32.EnterMonitorModeInDebugState()
// =====
// Take an exception in Debug state to Monitor mode.

AArch32.EnterMonitorModeInDebugState()
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = CurrentSecurityState() == SS_Secure;
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR_curr[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    PSTATE.E = SCTL.R.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTL.R.SPAN == '0' then
            PSTATE.PAN = '1';
    if HaveSSBSEExt() then PSTATE.SSBS = bit UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    if Havev8p9Debug() then
        DSPSR2 = bits(32) UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields(); // Update EDSCR processor state flags.

    EndOfInstruction();
```

## Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointByteMatch

```
// AArch32.WatchpointByteMatch()
// =====

boolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)
    integer top = 31;
    bottom = if DBGWVR[n]<2> == '1' then 2 else 3; // Word or doubleword
    byte_select_match = (DBGWCR[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR[n].MASK);

    // If DBGWCR[n].MASK is a nonzero value and DBGWCR[n].BAS is not set to '11111111', or
    // DBGWCR[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
    else
        LSB = (DBGWCR[n].BAS AND NOT(DBGWCR[n].BAS - 1)); MSB = (DBGWCR[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
            bottom = 3; // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        Constraint c;
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED return FALSE; // Disabled
            when Constraint_NONE mask = 0; // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    boolean WVR_match;
    if mask > bottom then
        WVR_match = (vaddress<top:mask> == DBGWVR[n]<top:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
    else
        WVR_match = vaddress<top:bottom> == DBGWVR[n]<top:bottom>;

    return (WVR_match && byte_select_match);
```

## Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointMatch

```
// AArch32.WatchpointMatch()
// =====
// Watchpoint matching in an AArch32 translation regime.

boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size,
                                AccessDescriptor accdesc)
    assert ELUsingAArch32\(S1TranslationRegime\(\)\);
    assert n < NumWatchpointsImplemented\(\);

    boolean enabled      = DBGWCR[n].E == '1';
    linked = DBGWCR[n].WT == '1';
    isbreakpnt = FALSE;
    linked_n = UInt(DBGWCR_EL1[n].LBN);
    state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC,
                                     linked, linked_n, isbreakpnt, accdesc);

    boolean ls_match;
    case DBGWCR[n].LSC<1:0> of
        when '00' ls_match = FALSE;
        when '01' ls_match = accdesc.read;
        when '10' ls_match = accdesc.write || accdesc.acctype == AccessType\_DC;
        when '11' ls_match = TRUE;

    boolean value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.Abort

```
// AArch32.Abort()
// =====
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32\(EL1\);

    if !route_to_aarch64 && EL2Enabled\(\) && !ELUsingAArch32\(EL2\) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                           (HaveRASExt\(\) && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
                           (IsDebugException(fault) && MDCR_EL2.TDE == '1'));

    if !route_to_aarch64 && HaveEL\(EL3\) && !ELUsingAArch32\(EL3\) then
        route_to_aarch64 = EffectiveEA\(\) == '1' && IsExternalAbort(fault);

    if route_to_aarch64 then
        AArch64.Abort(ZeroExtend(vaddress, 64), fault);
    elsif fault.accessdesc.acctype == AccessType\_IFETCH then
        AArch32.TakePrefetchAbortException(vaddress, fault);
    else
        AArch32.TakeDataAbortException(vaddress, fault);
```



## Library pseudocode for aarch32/exceptions/aborts/AArch32.AbortSyndrome

```
// AArch32.AbortSyndrome()
// =====
// Creates an exception syndrome record for Abort exceptions
// taken to Hyp mode
// from an AArch32 translation regime.

ExceptionRecord AArch32.AbortSyndrome(Exception exceptype, FaultRecord fault,
                                       bits(32) vaddress, bits(2) target_el)
    except = ExceptionSyndrome(exceptype);

    except.syndrome = AArch32.FaultSyndrome(exceptype, fault);
    except.vaddress = ZeroExtend(vaddress, 64);

    if IPAValid(fault) then
        except.ipavalid = TRUE;
        except.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        except.ipaddress = ZeroExtend(fault.ipaddress.address, 56);
    else
        except.ipavalid = FALSE;

    return except;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.CheckPCAlignment

```
// AArch32.CheckPCAlignment()
// =====

AArch32.CheckPCAlignment()
    bits(32) pc = ThisInstrAddr(32);

    if (CurrentInstrSet() == InstrSet_A32 && pc<1> == '1') || pc<0> == '1' then
        if AArch32.GeneralExceptionsToAArch64() then AArch64.PCAlignmentFault();

        AccessDescriptor accdesc = CreateAccDescIFetch();
        FaultRecord fault = NoFault(accdesc);
        // Generate an Alignment fault Prefetch Abort exception
        fault.statuscode = Fault_Alignment;
        AArch32.Abort(pc, fault);
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.CommonFaultStatus

```
// AArch32.CommonFaultStatus()
// =====
// Return the common part of the fault status on reporting a Data
// or Prefetch Abort.

bits(32) AArch32.CommonFaultStatus(FaultRecord fault, boolean long_format)
    bits(32) target = Zeros(32);
    if HaveRASExt() && IsAsyncAbort(fault) then
        ErrorState errstate = AArch32.PEErrorState(fault);
        target<15:14> = AArch32.EncodeAsyncErrorSyndrome(errstate); // AET
    if IsExternalAbort(fault) then target<12> = fault.extflag; // ExT
    target<9> = if long_format then '1' else '0'; // LPAE
    if long_format then // Long-descriptor format
        target<5:0> = EncodeLDFSC(fault.statuscode, fault.level); // STATUS
    else // Short-descriptor format
        target<10,3:0> = EncodeSDFSC(fault.statuscode, fault.level); // FS
    return target;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportDataAbort

```
// AArch32.ReportDataAbort()
// =====
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault,
                        bits(32) vaddress)
    boolean long_format;
    if route_to_monitor && CurrentSecurityState() != SS\_Secure then
        long_format = ((TTBCR_S.EAE == '1') ||
                        (IsExternalSyncAbort(fault) && ((PSTATE.EL == EL2 || TTBCR.EAE == '1') ||
                        (fault.secondstage && (boolean IMPLEMENTATION_DEFINED
                        "Report abort using Long-descriptor format")))));
    else
        long_format = TTBCR.EAE == '1';
    bits(32) syndrome = AArch32.CommonFaultStatus(fault, long_format);

    // bits of syndrome that are not common to I and D side
    if fault.accessdesc.acctype IN {AccessType\_DC, AccessType\_IC, AccessType\_AT} then
        syndrome<13> = '1'; // CM
        syndrome<11> = '1'; // WnR
    else
        syndrome<11> = if fault.write then '1' else '0'; // WnR

    if !long_format then
        syndrome<7:4> = fault.domain; // Domain

    if fault.accessdesc.acctype == AccessType\_IC then
        bits(32) i_syndrome;
        if (!long_format &&
            boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR") then
            i_syndrome = syndrome;
            syndrome<10,3:0> = EncodeSDFSC(Fault\_ICacheMaint, 1);
        else
            i_syndrome = bits(32) UNKNOWN;
        if route_to_monitor then
            IFSR_S = i_syndrome;
        else
            IFSR = i_syndrome;

    if route_to_monitor then
        DFSR_S = syndrome;
        DFAR_S = vaddress;
    else
        DFSR = syndrome;
        DFAR = vaddress;

    return;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort

```
// AArch32.ReportPrefetchAbort()
// =====
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)
    // The encoding used in the IFSR can be Long-descriptor format or Short-descriptor format.
    // Normally, the current translation table format determines the format. For an abort from
    // Non-secure state to Monitor mode, the IFSR uses the Long-descriptor format if any of the
    // following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * It is taken from Hyp mode.
    // * It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    long_format = FALSE;
    if route_to_monitor && CurrentSecurityState() != SS\_Secure then
        long_format = TTBCR_S.EAE == '1' || PSTATE.EL == EL2 || TTBCR.EAE == '1';
    else
        long_format = TTBCR.EAE == '1';

    bits(32) fsr = AArch32.CommonFaultStatus(fault, long_format);

    if route_to_monitor then
        IFSR_S = fsr;
        IFAR_S = vaddress;
    else
        IFSR = fsr;
        IFAR = vaddress;

    return;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.TakeDataAbortException

```
// AArch32.TakeDataAbortException()
// =====

AArch32.TakeDataAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && EffectiveEA() == '1' && IsExternalAbort(fault);
    route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
        (HCR.TGE == '1' ||
        (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
        (IsDebugException(fault) && HDCR.TDE == '1') ||
        IsSecondStage(fault)));

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x10;
    lr_offset = 8;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        except = AArch32.AbortSyndrome(Exception\_DataAbort, fault, vaddress, EL2);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32\_Abort, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException

```
// AArch32.TakePrefetchAbortException()
// =====

AArch32.TakePrefetchAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && EffectiveEA() == '1' && IsExternalAbort(fault);

    route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
        (HCR.TGE == '1' ||
            (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
            (IsDebugException(fault) && HDCR.TDE == '1') ||
            IsSecondStage(fault)));

    ExceptionRecord except;
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x0C;
    lr_offset = 4;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        if fault.statuscode == Fault Alignment then // PC Alignment fault
            except = ExceptionSyndrome(Exception PCAlignment);
            except.vaddress = ThisInstrAddr(64);
        else
            except = AArch32.AbortSyndrome(Exception InstructionAbort, fault, vaddress, EL2);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
    else
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32 Abort, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalFIQException

```
// AArch32.TakePhysicalFIQException()
// =====

AArch32.TakePhysicalFIQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.FMO == '1' && !IsInHost());

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.FIQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalFIQException();
    route_to_monitor = HaveEL(EL3) && SCR.FIQ == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR.TGE == '1' || HCR.FMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x1C;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        except = ExceptionSyndrome(Exception FIQ);
        AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32 FIQ, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalIRQException

```
// AArch32.TakePhysicalIRQException()
// =====
// Take an enabled physical IRQ exception.

AArch32.TakePhysicalIRQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32\(EL1\);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32\(EL2\) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.IMO == '1' && !IsInHost\(\));
    if !route_to_aarch64 && HaveEL\(EL3\) && !ELUsingAArch32\(EL3\) then
        route_to_aarch64 = SCR_EL3.IRQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalIRQException\(\);

    route_to_monitor = HaveEL\(EL3\) && SCR.IRQ == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR.TGE == '1' || HCR.IMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x18;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        except = ExceptionSyndrome\(Exception\_IRQ\);
        AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode\(M32\_IRQ, preferred\_exception\_return, lr\_offset, vect\_offset\);
```

## Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalErrorException

```
// AArch32.TakePhysicalErrorException()
// =====

AArch32.TakePhysicalErrorException(boolean implicit_esb)
    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1'));
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = EffectiveEA() == '1';

    if route_to_aarch64 then
        AArch64.TakePhysicalErrorException(implicit_esb);

    route_to_monitor = HaveEL(EL3) && SCR.EA == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR.TGE == '1' || HCR.AMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x10;
    lr_offset = 8;

    bits(2) target_el;
    if route_to_monitor then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_hyp then
        target_el = EL2;
    else
        target_el = EL1;

    FaultRecord fault = GetPendingPhysicalError();
    vaddress = bits(32) UNKNOWN;
    except = AArch32.AbortSyndrome(Exception\_DataAbort, fault, vaddress, target_el);

    if IsSErrorEdgeTriggered() then
        ClearPendingPhysicalError();
    case target_el of
        when EL3
            AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
            AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
        when EL2
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
        when EL1
            AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
            AArch32.EnterMode(M32\_Abort, preferred_exception_return, lr_offset, vect_offset);
    otherwise
        Unreachable();
```

## Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualFIQException

```
// AArch32.TakeVirtualFIQException()
// =====

AArch32.TakeVirtualFIQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then // Virtual IRQ enabled if TGE==0 and FMO==1
        assert HCR.TGE == '0' && HCR.FMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualFIQException();

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x1C;
    lr_offset = 4;

    AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualIRQException

```
// AArch32.TakeVirtualIRQException()
// =====

AArch32.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    if ELUsingAArch32(EL2) then // Virtual IRQs enabled if TGE==0 and IMO==1
        assert HCR.TGE == '0' && HCR.IMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualIRQException();

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x18;
    lr_offset = 4;

    AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualErrorException

```
// AArch32.TakeVirtualErrorException()
// =====

AArch32.TakeVirtualErrorException()

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then // Virtual SError enabled if TGE==0 and AMO==1
        assert HCR.TGE == '0' && HCR.AMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualErrorException();
    route_to_monitor = FALSE;

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x10;
    lr_offset = 8;

    vaddress = bits(32) UNKNOWN;
    parity = FALSE;
    Fault fault = Fault_AsyncExternal;
    integer level = integer UNKNOWN;
    bits(32) fsr = Zeros(32);
    if HaveRASExt() then
        if ELUsingAArch32(EL2) then
            fsr<15:14> = VDFSR.AET;
            fsr<12> = VDFSR.ExT;
        else
            fsr<15:14> = VESR_EL2.AET;
            fsr<12> = VESR_EL2.ExT;
    else
        fsr<12> = bit IMPLEMENTATION_DEFINED "Virtual External abort type";
    if TTBCR.EAE == '1' then // Long-descriptor format
        fsr<9> = '1';
        fsr<5:0> = EncodeLDFSC(fault, level);
    else // Short-descriptor format
        fsr<9> = '0';
        fsr<10,3:0> = EncodeSDFSC(fault, level);
    DFSR = fsr;
    DFAR = bits(32) UNKNOWN;
    ClearPendingVirtualSError();
    AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/debug/AArch32.SoftwareBreakpoint

```
// AArch32.SoftwareBreakpoint()
// =====

AArch32.SoftwareBreakpoint(bits(16) immediate)

    if (EL2Enabled() && !ELUsingAArch32(EL2) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')) || !ELUsingAArch32(EL1) then
        AArch64.SoftwareBreakpoint(immediate);

    accdesc = CreateAccDescIFetch();
    fault = NoFault(accdesc);
    vaddress = bits(32) UNKNOWN;

    fault.statuscode = Fault_Debug;
    fault.debugmoe = DebugException_BKPT;

    AArch32.Abort(vaddress, fault);
```



## Library pseudocode for aarch32/exceptions/debug/DebugException

```
constant bits(4) DebugException_Breakpoint = '0001';
constant bits(4) DebugException_BKPT      = '0011';
constant bits(4) DebugException_VectorCatch = '0101';
constant bits(4) DebugException_Watchpoint = '1010';
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.CheckAdvSIMDOrFPRegisterTraps

```
// AArch32.CheckAdvSIMDOrFPRegisterTraps()
// =====
// Check if an instruction that accesses an Advanced SIMD and
// floating-point System register is trapped by an appropriate HCR.TIDx
// ID group trap control.

AArch32.CheckAdvSIMDOrFPRegisterTraps(bits(4) reg)

    if PSTATE.EL == EL1 && EL2Enabled() then
        tid0 = if ELUsingAArch32(EL2) then HCR.TID0 else HCR_EL2.TID0;
        tid3 = if ELUsingAArch32(EL2) then HCR.TID3 else HCR_EL2.TID3;

        if ((tid0 == '1' && reg == '0000') ||
            (tid3 == '1' && reg IN {'0101', '0110', '0111'})) then // FPSID
            if ELUsingAArch32(EL2) then // MVFRx
                AArch32.SystemAccessTrap(M32\_Hyp, 0x8);
            else
                AArch64.AArch32SystemAccessTrap(EL2, 0x8);
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.ExceptionClass

```
// AArch32.ExceptionClass()
// =====
// Returns the Exception Class and Instruction Length fields to be reported in HSR

(integer,bit) AArch32.ExceptionClass(Exception exceptype)

    il_is_valid = TRUE;
    integer ec;
    case exceptype of
        when Exception Uncategorized          ec = 0x00; il_is_valid = FALSE;
        when Exception WFxTrap               ec = 0x01;
        when Exception CP15RRTTrap           ec = 0x03;
        when Exception CP15RRTTrap           ec = 0x04;
        when Exception CP14RTTrap            ec = 0x05;
        when Exception CP14DTTrap            ec = 0x06;
        when Exception AdvSIMDFPAccessTrap   ec = 0x07;
        when Exception FPIDTrap              ec = 0x08;
        when Exception PACTrap                ec = 0x09;
        when Exception TSTARTAccessTrap      ec = 0x1B;
        when Exception GPC                   ec = 0x1E;
        when Exception CP14RRTTrap           ec = 0x0C;
        when Exception BranchTarget          ec = 0x0D;
        when Exception IllegalState          ec = 0x0E; il_is_valid = FALSE;
        when Exception SupervisorCall        ec = 0x11;
        when Exception HypervisorCall        ec = 0x12;
        when Exception MonitorCall           ec = 0x13;
        when Exception InstructionAbort       ec = 0x20; il_is_valid = FALSE;
        when Exception PCAlignment           ec = 0x22; il_is_valid = FALSE;
        when Exception DataAbort             ec = 0x24;
        when Exception NV2DataAbort          ec = 0x25;
        when Exception FPTrappedException    ec = 0x28;
        when Exception PMU                   ec = 0x3D;
        otherwise                            Unreachable();

    if ec IN {0x20,0x24} && PSTATE.EL == EL2 then
        ec = ec + 1;
    bit il;
    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';

    return (ec,il);
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.GeneralExceptionsToAArch64

```
// AArch32.GeneralExceptionsToAArch64()
// =====
// Returns TRUE if exceptions normally routed to EL1 are being handled at an Exception
// level using AArch64, because either EL1 is using AArch64 or TGE is in force and EL2
// is using AArch64.

boolean AArch32.GeneralExceptionsToAArch64()
    return ((PSTATE.EL == EL0 && !ELUsingAArch32(EL1)) ||
            (EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1'));
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.ReportHypEntry

```
// AArch32.ReportHypEntry()
// =====
// Report syndrome information to Hyp mode registers.

AArch32.ReportHypEntry(ExceptionRecord except)

    Exception excepttype = except.excepttype;

    (ec,il) = AArch32.ExceptionClass(excepttype);
    iss = except.syndrome;
    iss2 = except.syndrome2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    HSR = ec<5:0>:il:iss;

    if excepttype IN {Exception_InstructionAbort, Exception_PCAlignment} then
        HIFAR = except.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elseif excepttype == Exception_DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = except.vaddress<31:0>;

    if except.ipavalid then
        HPFAR<31:4> = except.ipaddress<39:12>;
    else
        HPFAR<31:4> = bits(28) UNKNOWN;

    return;
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.ResetControlRegisters

```
// AArch32.ResetControlRegisters()
// =====
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.

AArch32.ResetControlRegisters(boolean cold_reset);
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.TakeReset

```
// AArch32.TakeReset()
// =====
// Reset into AArch32 state

AArch32.TakeReset(boolean cold_reset)
    assert !HaveAArch64();

    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL(EL3) then
        AArch32.WriteMode(M32_Svc);
        SCR.NS = '0'; // Secure state
    elseif HaveEL(EL2) then
        AArch32.WriteMode(M32_Hyp);
    else
        AArch32.WriteMode(M32_Svc);

    // Reset System registers in the coproc=0b111x encoding space
    // and other system components
    AArch32.ResetControlRegisters(cold_reset);
    FPEXC.EN = '0';

    // Reset all other PSTATE fields, including instruction set and endianness according to the
    // SCTLR values produced by the above call to ResetControlRegisters()
    PSTATE.<A,I,F> = '111'; // All asynchronous exceptions masked
    PSTATE.IT = '00000000'; // IT block state reset
    if HaveEL(EL2) && !HaveEL(EL3) then
        PSTATE.T = HSCTLR.TE; // Instruction set: TE=0:A32, TE=1:T32. PSTATE.J is RES0.
        PSTATE.E = HSCTLR.EE; // Endianness: EE=0: little-endian, EE=1: big-endian.
    else
        PSTATE.T = SCTLR.TE; // Instruction set: TE=0:A32, TE=1:T32. PSTATE.J is RES0.
        PSTATE.E = SCTLR.EE; // Endianness: EE=0: little-endian, EE=1: big-endian.
    PSTATE.IL = '0'; // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch32.ResetGeneralRegisters();
    AArch32.ResetSIMDFPRegisters();
    AArch32.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(32) rv; // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        if MVBAR<0> == '1' then // Reset vector in MVBAR
            rv = MVBAR<31:1>:'0';
        else
            rv = bits(32) IMPLEMENTATION_DEFINED "reset vector address";
    else
        rv = RVBAR<31:1>:'0';

    // The reset vector must be correctly aligned
    assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');

    boolean branch_conditional = FALSE;
    EDPRSR.R = '0'; // Leaving Reset State.
    BranchTo(rv, BranchType_RESET, branch_conditional);
```

### Library pseudocode for aarch32/exceptions/exceptions/ExcVectorBase

```
// ExcVectorBase()
// =====

bits(32) ExcVectorBase()
    if SCTLR.V == '1' then // Hivecs selected, base = 0xFFFF0000
        return Ones(16):Zeros(16);
    else
        return VBAR<31:5>:Zeros(5);
```

### Library pseudocode for aarch32/exceptions/ieeeefp/AArch32.FPTrappedException

```
// AArch32.FPTrappedException()
// =====

AArch32.FPTrappedException(bits(8) accumulated_exceptions)
    if AArch32.GeneralExceptionsToAArch64() then
        is_ase = FALSE;
        element = 0;
        AArch64.FPTrappedException(is_ase, accumulated_exceptions);
    FPEXC.DEX = '1';
    FPEXC.TFV = '1';
    FPEXC<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
    FPEXC<10:8> = '111'; // VECITR is RES1

    AArch32.TakeUndefInstrException();
```

### Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallHypervisor

```
// AArch32.CallHypervisor()
// =====
// Performs a HVC call

AArch32.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    if !ELUsingAArch32(EL2) then
        AArch64.CallHypervisor(immediate);
    else
        AArch32.TakeHVCEXception(immediate);
```

### Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallSupervisor

```
// AArch32.CallSupervisor()
// =====
// Calls the Supervisor

AArch32.CallSupervisor(bits(16) immediate_in)
    bits(16) immediate = immediate_in;
    if AArch32.CurrentCond() != '1110' then
        immediate = bits(16) UNKNOWN;
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CallSupervisor(immediate);
    else
        AArch32.TakeSVCEXception(immediate);
```

## Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeHVCEException

```
// AArch32.TakeHVCEException()
// =====

AArch32.TakeHVCEException(bits(16) immediate)
    assert HaveEL(EL2) && ELUsingAArch32(EL2);

    AArch32.ITAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr(32);
    vect_offset = 0x08;

    except = ExceptionSyndrome(Exception_HypervisorCall);
    except.syndrome<15:0> = immediate;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
```

## Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSMCEException

```
// AArch32.TakeSMCEException()
// =====

AArch32.TakeSMCEException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    AArch32.ITAdvance();
    HSAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr(32);
    vect_offset = 0x08;
    lr_offset = 0;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSVCEException

```
// AArch32.TakeSVCEException()
// =====

AArch32.TakeSVCEException(bits(16) immediate)

    AArch32.ITAdvance();
    SSAdvance();
    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

    bits(32) preferred_exception_return = NextInstrAddr(32);
    vect_offset = 0x08;
    lr_offset = 0;

    if PSTATE.EL == EL2 || route_to_hyp then
        except = ExceptionSyndrome(Exception_SupervisorCall);
        except.syndrome<15:0> = immediate;
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterHypMode

```
// AArch32.EnterHypMode()
// =====
// Take an exception to Hyp mode.

AArch32.EnterHypMode(ExceptionRecord except, bits(32) preferred_exception_return,
                    integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    if Halted() then
        AArch32.EnterHypModeInDebugState(except);
        return;
    bits(32) spsr = GetPSRFromPSTATE(AArch32.NonDebugState, 32);
    if !(except.excepttype IN {Exception_IRQ, Exception_FIQ}) then
        AArch32.ReportHypEntry(except);
    AArch32.WriteMode(M32_Hyp);
    SPSR_curr[] = spsr;
    ELR_hyp = preferred_exception_return;
    PSTATE.T = HSCTLR.TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    if !HaveEL(EL3) || SCR_curr[].EA == '0' then PSTATE.A = '1';
    if !HaveEL(EL3) || SCR_curr[].IRQ == '0' then PSTATE.I = '1';
    if !HaveEL(EL3) || SCR_curr[].FIQ == '0' then PSTATE.F = '1';
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HaveSSBSExt() then PSTATE.SSBS = HSCTLR.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(HVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE); // Check for debug event on exception entry

    EndOfInstruction();
```

## Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMode

```
// AArch32.EnterMode()
// =====
// Take an exception to a mode other than Monitor and Hyp mode.

AArch32.EnterMode(bits(5) target_mode, bits(32) preferred_exception_return, integer lr_offset,
                  integer vect_offset)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if Halted() then
        AArch32.EnterModeInDebugState(target_mode);
        return;
    bits(32) spsr = GetPSRFromPSTATE(AArch32.NonDebugState, 32);
    if PSTATE.M == M32\_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR\_curr[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTL.R.TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    if target_mode == M32\_FIQ then
        PSTATE.<A,I,F> = '111';
    elsif target_mode IN {M32\_Abort, M32\_IRQ} then
        PSTATE.<A,I> = '11';
    else
        PSTATE.I = '1';
    PSTATE.E = SCTL.R.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTL.R.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = SCTL.R.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(ExcVectorBase()<31:5>:vect_offset<4:0>, BranchType\_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE); // Check for debug event on exception entry

    EndOfInstruction();
```



## Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMonitorMode

```
// AArch32.EnterMonitorMode()
// =====
// Take an exception to Monitor mode.

AArch32.EnterMonitorMode(bits(32) preferred_exception_return, integer lr_offset,
                          integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = CurrentSecurityState() == SS_Secure;
    if Halted() then
        AArch32.EnterMonitorModeInDebugState();
        return;
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState, 32);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR_curr[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTL.R.TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    PSTATE.<A,I,F> = '111';
    PSTATE.E = SCTL.R.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elseif SCTL.R.SPAN == '0' then
            PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = SCTL.R.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(MVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE); // Check for debug event on exception entry

    EndOfInstruction();
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckAdvSIMDOrFPEEnabled

```
// AArch32.CheckAdvSIMDOrFPEEnabled()
// =====
// Check against CPACR, FPEXC, HCPTR, NSACR, and CPTR_EL3.

AArch32.CheckAdvSIMDOrFPEEnabled(boolean fpexc_check_in, boolean advsimd)
    boolean fpexc_check = fpexc_check_in;
    if PSTATE.EL == EL0 && !ELUsingAArch32\(EL1\) then
        // When executing at EL0 using AArch32, if EL1 is using AArch64 then the Effective value of
        // FPEXC.EN is 1. This includes when EL2 is using AArch64 and enabled in the current
        // Security state, HCR_EL2.TGE is 1, and the Effective value of HCR_EL2.RW is 1.
        AArch64.CheckFPAdvSIMDEnabled\(\);
    else
        cpacr_asedis = CPACR.ASEDIS;
        cpacr_cp10 = CPACR.cp10;

        if HaveEL\(EL3\) && ELUsingAArch32\(EL3\) && CurrentSecurityState\(\) == SS NonSecure then
            // Check if access disabled in NSACR
            if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
            if NSACR.cp10 == '0' then cpacr_cp10 = '00';

        if PSTATE.EL != EL2 then
            // Check if Advanced SIMD disabled in CPACR
            if advsimd && cpacr_asedis == '1' then AArch32.Undefined\(\);

            // Check if access disabled in CPACR
            boolean disabled;
            case cpacr_cp10 of
                when '00' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0;
                when '10' disabled = ConstrainUnpredictableBool\(Unpredictable\_RESCPACR\);
                when '11' disabled = FALSE;
            if disabled then AArch32.Undefined\(\);

        // If required, check FPEXC enabled bit.
        if (fpexc_check && PSTATE.EL == EL0 && EL2Enabled\(\) && !ELUsingAArch32\(EL2\) &&
            HCR_EL2.TGE == '1') then
            // When executing at EL0 using AArch32, if EL2 is using AArch64 and enabled in the
            // current Security state, HCR_EL2.TGE is 1, and the Effective value of HCR_EL2.RW is 0
            // then it is IMPLEMENTATION DEFINED whether the Effective value of FPEXC.EN is 1
            // or the value of FPEXC32_EL2.EN.
            fpexc_check = (boolean IMPLEMENTATION_DEFINED
                "Use FPEXC32_EL2.EN value when {TGE,RW} == {1,0}");

        if fpexc_check && FPEXC.EN == '0' then
            AArch32.Undefined\(\);

        AArch32.CheckFPAdvSIMDTrap(advsimd);    // Also check against HCPTR and CPTR_EL3
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap

```
// AArch32.CheckFPAdvSIMDTrap()
// =====
// Check against CPTR_EL2 and CPTR_EL3.

AArch32.CheckFPAdvSIMDTrap(boolean advsimd)
    if EL2Enabled\(\) && !ELUsingAArch32\(EL2\) then
        AArch64.CheckFPAdvSIMDTrap\(\);
    else
        if (HaveEL\(EL3\) && !ELUsingAArch32\(EL3\) &&
            CPTR_EL3.TFP == '1' && EL3SDDUndefPriority\(\)) then
            UNDEFINED;

        ss = CurrentSecurityState\(\);
        if HaveEL\(EL2\) && ss != SS\_Secure then
            hcptr_tase = HCPTR.TASE;
            hcptr_cp10 = HCPTR.TCP10;

            if HaveEL\(EL3\) && ELUsingAArch32\(EL3\) then
                // Check if access disabled in NSACR
                if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
                if NSACR.cp10 == '0' then hcptr_cp10 = '1';

            // Check if access disabled in HCPTR
            if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
                except = ExceptionSyndrome\(Exception\_AdvSIMDFPAccessTrap\);
                except.syndrome<24:20> = ConditionSyndrome\(\);

                if advsimd then
                    except.syndrome<5> = '1';
                else
                    except.syndrome<5> = '0';
                    except.syndrome<3:0> = '1010';           // coproc field, always 0xA

                if PSTATE.EL == EL2 then
                    AArch32.TakeUndefInstrException(except);
                else
                    AArch32.TakeHypTrapException(except);

            if HaveEL\(EL3\) && !ELUsingAArch32\(EL3\) then
                // Check if access disabled in CPTR_EL3
                if CPTR_EL3.TFP == '1' then
                    if EL3SDDUndef\(\) then
                        UNDEFINED;
                    else
                        AArch64.AdvSIMDFPAccessTrap\(EL3\);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSMCUndefOrTrap

```
// AArch32.CheckForSMCUndefOrTrap()
// =====
// Check for UNDEFINED or trap on SMC instruction

AArch32.CheckForSMCUndefOrTrap()
    if !HaveEL\(EL3\) || PSTATE.EL == EL0 then
        UNDEFINED;

    if EL2Enabled\(\) && !ELUsingAArch32\(EL2\) then
        AArch64.CheckForSMCUndefOrTrap\(Zeros\(16\)\);
    else
        route_to_hyp = EL2Enabled\(\) && PSTATE.EL == EL1 && HCR.TSC == '1';
        if route_to_hyp then
            except = ExceptionSyndrome\(Exception\_MonitorCall\);
            AArch32.TakeHypTrapException(except);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSVCTrap

```
// AArch32.CheckForSVCTrap()
// =====
// Check for trap on SVC instruction

AArch32.CheckForSVCTrap(bits(16) immediate)
    if HaveFGTExt() then
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
                (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));

        if route_to_el2 then
            except = ExceptionSyndrome(Exception_SupervisorCall);
            except.syndrome<15:0> = immediate;
            except.trappedsyscallinst = TRUE;
            bits(64) preferred_exception_return = ThisInstrAddr(64);
            vect_offset = 0x0;

            AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForWFXTrap

```
// AArch32.CheckForWFXTrap()
// =====
// Check for trap on WFE or WFI instruction

AArch32.CheckForWFXTrap(bits(2) target_el, WFXType wfxtype)
    assert HaveEL(target_el);

    // Check for routing to AArch64
    if !ELUsingAArch32(target_el) then
        AArch64.CheckForWFXTrap(target_el, wfxtype);
        return;

    boolean is_wfe = wfxtype == WFXType_WFE;
    boolean trap;
    case target_el of
        when EL1
            trap = (if is_wfe then SCTLR.nTWE else SCTLR.nTWI) == '0';
        when EL2
            trap = (if is_wfe then HCR.TWE else HCR.TWI) == '1';
        when EL3
            trap = (if is_wfe then SCR.TWE else SCR.TWI) == '1';

    if trap then
        if target_el == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.WFXTrap(wfxtype, target_el);

        if target_el == EL3 then
            AArch32.TakeMonitorTrapException();
        elsif target_el == EL2 then
            except = ExceptionSyndrome(Exception_WFXTrap);
            except.syndrome<24:20> = ConditionSyndrome();

            case wfxtype of
                when WFXType_WFI
                    except.syndrome<0> = '0';
                when WFXType_WFE
                    except.syndrome<0> = '1';

            AArch32.TakeHypTrapException(except);
        else
            AArch32.TakeUndefInstrException();
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckITEnabled

```
// AArch32.CheckITEnabled()
// =====
// Check whether the T32 IT instruction is disabled.

AArch32.CheckITEnabled(bits(4) mask)
    bit it_disabled;
    if PSTATE.EL == EL2 then
        it_disabled = HSCTLR.ITD;
    else
        it_disabled = (if ELUsingAArch32(EL1) then SCTLR.ITD else SCTLR_ELx[].ITD);
    if it_disabled == '1' then
        if mask != '1000' then UNDEFINED;

    accdesc = CreateAccDescIFetch();
    aligned = TRUE;
    // Otherwise whether the IT block is allowed depends on hwl of the next instruction.
    next_instr = AArch32.MemSingle(NextInstrAddr(32), 2, accdesc, aligned);

    if next_instr IN {'11xxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxxx',
                     '01001xxxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'} then
        // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
        // taken on the IT instruction or the next instruction. This is not reflected in
        // the pseudocode, which always takes the exception on the IT instruction. This
        // also does not take into account cases where the next instruction is UNPREDICTABLE.
        UNDEFINED;

    return;
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckIllegalState

```
// AArch32.CheckIllegalState()
// =====
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch32.CheckIllegalState()
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CheckIllegalState();
    elsif PSTATE.IL == '1' then
        route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

        bits(32) preferred_exception_return = ThisInstrAddr(32);
        vect_offset = 0x04;

        if PSTATE.EL == EL2 || route_to_hyp then
            except = ExceptionSyndrome(Exception\_IllegalState);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
        else
            AArch32.TakeUndefInstrException();
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckSETENDEnabled

```
// AArch32.CheckSETENDEnabled()
// =====
// Check whether the AArch32 SETEND instruction is disabled.

AArch32.CheckSETENDEnabled()
    bit setend_disabled;
    if PSTATE.EL == EL2 then
        setend_disabled = HSCTLR.SED;
    else
        setend_disabled = (if ELUsingAArch32(EL1) then SCTLR.SED else SCTLR_ELx[].SED);
    if setend_disabled == '1' then
        UNDEFINED;

    return;
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.SystemAccessTrap

```
// AArch32.SystemAccessTrap()
// =====
// Trapped System register access.

AArch32.SystemAccessTrap(bits(5) mode, integer ec)
    (valid, target_el) = ELFromM32(mode);
    assert valid && HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    if target_el == EL2 then
        except = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
        AArch32.TakeHypTrapException(except);
    else
        AArch32.TakeUndefInstrException();
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.SystemAccessTrapSyndrome

```
// AArch32.SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS,
// VMSR instructions, other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch32.SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord except;

    case ec of
        when 0x0    except = ExceptionSyndrome(Exception Uncategorized);
        when 0x3    except = ExceptionSyndrome(Exception CP15RTTrap);
        when 0x4    except = ExceptionSyndrome(Exception CP15RRTTrap);
        when 0x5    except = ExceptionSyndrome(Exception CP14RTTrap);
        when 0x6    except = ExceptionSyndrome(Exception CP14DTTrap);
        when 0x7    except = ExceptionSyndrome(Exception AdvSIMDFPAccessTrap);
        when 0x8    except = ExceptionSyndrome(Exception FPIDTrap);
        when 0xC    except = ExceptionSyndrome(Exception CP14RRTTrap);
        otherwise   Unreachable();

    bits(20) iss = Zeros(20);

    if except.exceptype == Exception Uncategorized then
        return except;
    elseif except.exceptype IN {Exception FPIDTrap, Exception CP14RTTrap,
                               Exception CP15RTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        iss<13:10> = instr<19:16>;           // CRn, Reg in case of VMRS
        iss<8:5>   = instr<15:12>;           // Rt
        iss<9>     = '0';                     // RES0

        if except.exceptype != Exception FPIDTrap then // When trap is not for VMRS
            iss<19:17> = instr<7:5>;           // opc2
            iss<16:14> = instr<23:21>;          // opc1
            iss<4:1>   = instr<3:0>;           // CRm
        else //VMRS Access
            iss<19:17> = '000';                 //opc2 - Hardcoded for VMRS
            iss<16:14> = '111';                 //opc1 - Hardcoded for VMRS
            iss<4:1>   = '0000';                //CRm - Hardcoded for VMRS
    elseif except.exceptype IN {Exception CP14RRTTrap, Exception AdvSIMDFPAccessTrap,
                               Exception CP15RRTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss<19:16> = instr<7:4>;           // opc1
        iss<13:10> = instr<19:16>;           // Rt2
        iss<8:5>   = instr<15:12>;           // Rt
        iss<4:1>   = instr<3:0>;           // CRm
    elseif except.exceptype == Exception CP14DTTrap then
        // Trapped LDC/STC
        iss<19:12> = instr<7:0>;           // imm8
        iss<4>     = instr<23>;             // U
        iss<2:1>   = instr<24,21>;          // P,W
        if instr<19:16> == '1111' then // Rn==15, LDC(Literal addressing)/STC
            iss<8:5> = bits(4) UNKNOWN;
            iss<3>   = '1';
        iss<0> = instr<20>;                 // Direction

    except.syndrome<24:20> = ConditionSyndrome();
    except.syndrome<19:0>  = iss;

    return except;
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.TakeHypTrapException

```
// AArch32.TakeHypTrapException()
// =====
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(integer ec)
    except = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch32.TakeHypTrapException(except);

// AArch32.TakeHypTrapException()
// =====
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(ExceptionRecord except)
    assert HaveEL(EL2) && CurrentSecurityState() == SS\_NonSecure && ELUsingAArch32(EL2);

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x14;

    AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.TakeMonitorTrapException

```
// AArch32.TakeMonitorTrapException()
// =====
// Exceptions routed to Monitor mode as a Monitor Trap exception.

AArch32.TakeMonitorTrapException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet\_A32 then 4 else 2;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.TakeUndefInstrException

```
// AArch32.TakeUndefInstrException()
// =====

AArch32.TakeUndefInstrException()
    except = ExceptionSyndrome(Exception Uncategorized);
    AArch32.TakeUndefInstrException(except);

// AArch32.TakeUndefInstrException()
// =====

AArch32.TakeUndefInstrException(ExceptionRecord except)

    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet\_A32 then 4 else 2;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
    elsif route_to_hyp then
        AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32\_Undef, preferred_exception_return, lr_offset, vect_offset);
```



## Library pseudocode for aarch32/exceptions/traps/AArch32.Undefined

```
// AArch32.Undefined()
// =====

AArch32.Undefined()

    if AArch32.GeneralExceptionsToAArch64() then AArch64.Undefined();
    AArch32.TakeUndefInstrException();
```

## Library pseudocode for aarch32/functions/aborts/AArch32.DomainValid

```
// AArch32.DomainValid()
// =====
// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

boolean AArch32.DomainValid(Fault statuscode, integer level)
    assert statuscode != Fault\_None;

    case statuscode of
        when Fault\_Domain
            return TRUE;
        when Fault\_Translation, Fault\_AccessFlag, Fault\_SyncExternalOnWalk, Fault\_SyncParityOnWalk
            return level == 2;
        otherwise
            return FALSE;
```

## Library pseudocode for aarch32/functions/aborts/AArch32.FaultSyndrome

```
// AArch32.FaultSyndrome()
// =====
// Creates an exception syndrome value and updates the virtual address for Abort and Watchpoint
// exceptions taken to AArch32 Hyp mode.

bits(25) AArch32.FaultSyndrome(Exception exceptype, FaultRecord fault)
    assert fault.statuscode != Fault\_None;

    bits(25) iss = Zeros(25);

    boolean d_side = exceptype == Exception\_DataAbort;
    if HaveRASExt() && IsAsyncAbort(fault) then
        ErrorState errstate = AArch32.PEErrorState(fault);
        iss<11:10> = AArch32.EncodeAsyncErrorSyndrome(errstate); // AET

    if d_side then
        if (IsSecondStage(fault) && !fault.s2fslwalk &&
            (!IsExternalSyncAbort(fault) ||
            (!HaveRASExt() && fault.accessdesc.acctype == AccessType\_TTW &&
            boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk")))) then
            iss<24:14> = LSInstructionSyndrome();

        if fault.accessdesc.acctype IN {AccessType\_DC, AccessType\_IC, AccessType\_AT} then
            iss<8> = '1';

        if fault.accessdesc.acctype IN {AccessType\_DC, AccessType\_IC, AccessType\_AT} then
            iss<6> = '1';
        elsif fault.statuscode IN {Fault\_HWUpdateAccessFlag, Fault\_Exclusive} then
            iss<6> = bit UNKNOWN;
        elsif fault.accessdesc.atomicop && IsExternalAbort(fault) then
            iss<6> = bit UNKNOWN;
        else
            iss<6> = if fault.write then '1' else '0';

    if IsExternalAbort(fault) then iss<9> = fault.extflag;
    iss<7> = if fault.s2fslwalk then '1' else '0';
    iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return (iss);
```

## Library pseudocode for aarch32/functions/aborts/EncodeSDFSC

```
// EncodeSDFSC()
// =====
// Function that gives the Short-descriptor FSR code for different types of Fault

bits(5) EncodeSDFSC(Fault statuscode, integer level)

    bits(5) result;
    case statuscode of
        when Fault AccessFlag
            assert level IN {1,2};
            result = if level == 1 then '00011' else '00110';
        when Fault Alignment
            result = '00001';
        when Fault Permission
            assert level IN {1,2};
            result = if level == 1 then '01101' else '01111';
        when Fault Domain
            assert level IN {1,2};
            result = if level == 1 then '01001' else '01011';
        when Fault Translation
            assert level IN {1,2};
            result = if level == 1 then '00101' else '00111';
        when Fault SyncExternal
            result = '01000';
        when Fault SyncExternalOnWalk
            assert level IN {1,2};
            result = if level == 1 then '01100' else '01110';
        when Fault SyncParity
            result = '11001';
        when Fault SyncParityOnWalk
            assert level IN {1,2};
            result = if level == 1 then '11100' else '11110';
        when Fault AsyncParity
            result = '11000';
        when Fault AsyncExternal
            result = '10110';
        when Fault Debug
            result = '00010';
        when Fault TLBConflict
            result = '10000';
        when Fault Lockdown
            result = '10100';    // IMPLEMENTATION DEFINED
        when Fault Exclusive
            result = '10101';    // IMPLEMENTATION DEFINED
        when Fault ICacheMaint
            result = '00100';
        otherwise
            Unreachable();

    return result;
```

## Library pseudocode for aarch32/functions/common/A32ExpandImm

```
// A32ExpandImm()
// =====

bits(32) A32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = A32ExpandImm\_C(imm12, PSTATE.C);

    return imm32;
```

### Library pseudocode for aarch32/functions/common/A32ExpandImm\_C

```
// A32ExpandImm_C()
// =====

(bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)

    unrotated_value = ZeroExtend(imm12<7:0>, 32);
    (imm32, carry_out) = Shift_C(unrotated_value, SRTYPE_ROR, 2*UInt(imm12<11:8>), carry_in);

    return (imm32, carry_out);
```

### Library pseudocode for aarch32/functions/common/DecodeImmShift

```
// DecodeImmShift()
// =====

(SRTYPE, integer) DecodeImmShift(bits(2) srtype, bits(5) imm5)

    SRTYPE shift_t;
    integer shift_n;
    case srtype of
        when '00'
            shift_t = SRTYPE_LSL; shift_n = UInt(imm5);
        when '01'
            shift_t = SRTYPE_LSR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '10'
            shift_t = SRTYPE_ASR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '11'
            if imm5 == '00000' then
                shift_t = SRTYPE_RRX; shift_n = 1;
            else
                shift_t = SRTYPE_ROR; shift_n = UInt(imm5);

    return (shift_t, shift_n);
```

### Library pseudocode for aarch32/functions/common/DecodeRegShift

```
// DecodeRegShift()
// =====

SRTYPE DecodeRegShift(bits(2) srtype)

    SRTYPE shift_t;
    case srtype of
        when '00' shift_t = SRTYPE_LSL;
        when '01' shift_t = SRTYPE_LSR;
        when '10' shift_t = SRTYPE_ASR;
        when '11' shift_t = SRTYPE_ROR;

    return shift_t;
```

### Library pseudocode for aarch32/functions/common/RRX

```
// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)

    (result, -) = RRX_C(x, carry_in);
    return result;
```

## Library pseudocode for aarch32/functions/common/RRX\_C

```
// RRX_C()
// =====

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);
```

## Library pseudocode for aarch32/functions/common/SRType

```
// SRType
// =====

enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};
```

## Library pseudocode for aarch32/functions/common/Shift

```
// Shift()
// =====

bits(N) Shift(bits(N) value, SRType srtype, integer amount, bit carry_in)
    (result, -) = Shift\_C(value, srtype, amount, carry_in);
    return result;
```

## Library pseudocode for aarch32/functions/common/Shift\_C

```
// Shift_C()
// =====

(bits(N), bit) Shift_C(bits(N) value, SRType srtype, integer amount, bit carry_in)
    assert !(srtype == SRType\_RRX && amount != 1);

    bits(N) result;
    bit carry_out;
    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case srtype of
            when SRType\_LSL
                (result, carry_out) = LSL\_C(value, amount);
            when SRType\_LSR
                (result, carry_out) = LSR\_C(value, amount);
            when SRType\_ASR
                (result, carry_out) = ASR\_C(value, amount);
            when SRType\_ROR
                (result, carry_out) = ROR\_C(value, amount);
            when SRType\_RRX
                (result, carry_out) = RRX\_C(value, carry_in);

    return (result, carry_out);
```

## Library pseudocode for aarch32/functions/common/T32ExpandImm

```
// T32ExpandImm()
// =====

bits(32) T32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = T32ExpandImm\_C(imm12, PSTATE.C);

    return imm32;
```

### Library pseudocode for aarch32/functions/common/T32ExpandImm\_C

```
// T32ExpandImm_C()
// =====

(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)
    bits(32) imm32;
    bit carry_out;
    if imm12<11:10> == '00' then
        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
            when '10'
                imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
            when '11'
                imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
        carry_out = carry_in;
    else
        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

    return (imm32, carry_out);
```

### Library pseudocode for aarch32/functions/common/VBitOps

```
// VBitOps
// =====

enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};
```

### Library pseudocode for aarch32/functions/common/VCGEType

```
// VCGEType
// =====

enumeration VCGEType {VCGEType_signed, VCGEType_unsigned, VCGEType_fp};
```

### Library pseudocode for aarch32/functions/common/VCGTtype

```
// VCGTtype
// =====

enumeration VCGTtype {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};
```

### Library pseudocode for aarch32/functions/common/VFPNegMul

```
// VFPNegMul
// =====

enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};
```

## Library pseudocode for aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps

```
// AArch32.CheckCP15InstrCoarseTraps()
// =====
// Check for coarse-grained traps to System registers in the
// coproc=0b1111 encoding space by HSTR and HCR.

AArch32.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)
    if PSTATE.EL == EL0 && (!ELUsingAArch32\(EL1\) ||
        (EL2Enabled\(\) && !ELUsingAArch32\(EL2\))) then
        AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm);

    trapped_encoding = ((CRn == 9 && CRm IN {0,1,2, 5,6,7,8 }) ||
        (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
        (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled\(\) then
        major = if nreg == 1 then CRn else CRm;
        // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR<CRn/CRm>
        // and MRC and MCR disabled by HCR.TIDCP.
        if ((!(major IN {4,14}) && HSTR<major> == '1') ||
            (HCR.TIDCP == '1' && nreg == 1 && trapped_encoding)) then
            if (PSTATE.EL == EL0 &&
                boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0") then
                UNDEFINED;
            if ELUsingAArch32\(EL2\) then
                AArch32.SystemAccessTrap(M32_Hyp, 0x3);
            else
                AArch64.AArch32SystemAccessTrap(EL2, 0x3);
```

## Library pseudocode for aarch32/functions/exclusive/AArch32.ExclusiveMonitorsPass

```
// AArch32.ExclusiveMonitorsPass()
// =====
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

// It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
// before or after the check on the local Exclusives monitor. As a result a failure
// of the local monitor can occur on some implementations even if the memory
// access would give an memory abort.

boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)
    boolean acqrel = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescExLDST(MemOp\_STORE, acqrel, tagchecked);
    boolean aligned = IsAligned(address, size);

    if !aligned then
        AArch32.Abort(address, AlignmentFault(accdesc));

    if !AArch32.IsExclusiveVA(address, ProcessorID(), size) then
        return FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.address, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed && memaddrdesc.memattrs.shareability != Shareability\_NSH then
        passed = IsExclusiveGlobal(memaddrdesc.address, ProcessorID(), size);

    return passed;
```

## Library pseudocode for aarch32/functions/exclusive/AArch32.IsExclusiveVA

```
// AArch32.IsExclusiveVA()
// =====
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.

boolean AArch32.IsExclusiveVA(bits(32) address, integer processorid, integer size);
```

## Library pseudocode for aarch32/functions/exclusive/AArch32.MarkExclusiveVA

```
// AArch32.MarkExclusiveVA()
// =====
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.

AArch32.MarkExclusiveVA(bits(32) address, integer processorid, integer size);
```

## Library pseudocode for aarch32/functions/exclusive/AArch32.SetExclusiveMonitors

```
// AArch32.SetExclusiveMonitors()
// =====
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch32.SetExclusiveMonitors(bits(32) address, integer size)
    boolean acqrel = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescExLDST(MemOp\_LOAD, acqrel, tagchecked);
    boolean aligned = IsAligned(address, size);

    if !aligned then
        AArch32.Abort(address, AlignmentFault(accdesc));

    memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch32.MarkExclusiveVA(address, ProcessorID(), size);
```

## Library pseudocode for aarch32/functions/float/CheckAdvSIMDEnabled

```
// CheckAdvSIMDEnabled()
// =====

CheckAdvSIMDEnabled()

    fpexc_check = TRUE;
    advsimd = TRUE;

    AArch32.CheckAdvSIMDOrFPEEnabled(fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEEnabled() occurs only if Advanced SIMD access is permitted

    // Make temporary copy of D registers
    // _Dclone[] is used as input data for instruction pseudocode
    for i = 0 to 31
        _Dclone[i] = D[i];

    return;
```

## Library pseudocode for aarch32/functions/float/CheckAdvSIMDOrVFPEEnabled

```
// CheckAdvSIMDOrVFPEEnabled()
// =====

CheckAdvSIMDOrVFPEEnabled(boolean include_fpexc_check, boolean advsimd)
    AArch32.CheckAdvSIMDOrFPEEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEEnabled() occurs only if VFP access is permitted
    return;
```

## Library pseudocode for aarch32/functions/float/CheckCryptoEnabled32

```
// CheckCryptoEnabled32()
// =====

CheckCryptoEnabled32()
    CheckAdvSIMDEnabled();
    // Return from CheckAdvSIMDEnabled() occurs only if access is permitted
    return;
```

## Library pseudocode for aarch32/functions/float/CheckVFPEEnabled

```
// CheckVFPEEnabled()
// =====

CheckVFPEEnabled(boolean include_fpexc_check)
    advsimd = FALSE;
    AArch32.CheckAdvSIMDOrFPEEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEEnabled() occurs only if VFP access is permitted
    return;
```



## Library pseudocode for aarch32/functions/float/FPHalvedSub

```
// FPHalvedSub()
// =====

bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPTType Infinity);  inf2 = (type2 == FPTType Infinity);
        zero1 = (type1 == FPTType Zero);      zero2 = (type2 == FPTType Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 != sign2 then
            result = FPZero(sign1, N);
        else
            result_value = (value1 - value2) / 2.0;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, N);
    return result;
```

## Library pseudocode for aarch32/functions/float/FPRSqrtStep

```
// FPRSqrtStep()
// =====

bits(N) FPRSqrtStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRTType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPTType Infinity);  inf2 = (type2 == FPTType Infinity);
        zero1 = (type1 == FPTType Zero);      zero2 = (type2 == FPTType Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0', N);
        else
            product = FPMul(op1, op2, fpcr);
        bits(N) three = FPThree('0', N);
        result = FPHalvedSub(three, product, fpcr);
    return result;
```

## Library pseudocode for aarch32/functions/float/FPRecipStep

```
// FPRecipStep()
// =====

bits(N) FPRecipStep(bits(N) op1, bits(N) op2)
  assert N IN {16,32};
  FPCRTType fpcr = StandardFPSCRValue();
  (type1,sign1,value1) = FPUnpack(op1, fpcr);
  (type2,sign2,value2) = FPUnpack(op2, fpcr);
  (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
  if !done then
    inf1 = (type1 == FPTType\_Infinity);  inf2 = (type2 == FPTType\_Infinity);
    zero1 = (type1 == FPTType\_Zero);      zero2 = (type2 == FPTType\_Zero);
    bits(N) product;
    if (inf1 && zero2) || (zero1 && inf2) then
      product = FPZero('0', N);
    else
      product = FPMul(op1, op2, fpcr);
    bits(N) two = FPTwo('0', N);
    result = FPSub(two, product, fpcr);
  return result;
```

## Library pseudocode for aarch32/functions/float/StandardFPSCRValue

```
// StandardFPSCRValue()
// =====

FPCRTType StandardFPSCRValue()
  bits(32) value = '00000' : FPSCR.AHP : '110000' : FPSCR.FZ16 : '00000000000000000000';
  return ZeroExtend(value, 64);
```

## Library pseudocode for aarch32/functions/memory/AArch32.MemSingle

```
// AArch32.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccessDescriptor accdesc_in,
                                     boolean aligned]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    AccessDescriptor accdesc = accdesc_in;
    assert IsAligned(address, size);

    AddressDescriptor memaddrdesc;
    memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Memory array access
    if SPESampleInFlight then
        boolean is_load = TRUE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);

    PhysMemRetStatus memstatus;
    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
    return value;

// AArch32.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccessDescriptor accdesc_in,
                                     boolean aligned] = bits(size*8) value
    assert size IN {1, 2, 4, 8, 16};
    AccessDescriptor accdesc = accdesc_in;
    assert IsAligned(address, size);

    AddressDescriptor memaddrdesc;
    memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    if SPESampleInFlight then
        boolean is_load = FALSE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);

    PhysMemRetStatus memstatus;
    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
    return;
```

### Library pseudocode for aarch32/functions/memory/AArch32.UnalignedAccessFaults

```
// AArch32.UnalignedAccessFaults()
// =====
// Determine whether the unaligned access generates an Alignment fault

boolean AArch32.UnalignedAccessFaults(AccessDescriptor accdesc)
    return (AlignmentEnforced() ||
           accdesc.a32lsmd ||
           accdesc.exclusive ||
           accdesc.acqsc ||
           accdesc.relsc);
```

### Library pseudocode for aarch32/functions/memory/Hint\_PreloadData

```
// Hint_PreloadData()
// =====

Hint_PreloadData(bits(32) address);
```

### Library pseudocode for aarch32/functions/memory/Hint\_PreloadDataForWrite

```
// Hint_PreloadDataForWrite()
// =====

Hint_PreloadDataForWrite(bits(32) address);
```

### Library pseudocode for aarch32/functions/memory/Hint\_PreloadInstr

```
// Hint_PreloadInstr()
// =====

Hint_PreloadInstr(bits(32) address);
```

### Library pseudocode for aarch32/functions/memory/MemA

```
// MemA[] - non-assignment form
// =====

bits(8*size) MemA(bits(32) address, integer size)
    boolean acqrel = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescExLDST(MemOp\_LOAD, acqrel, tagchecked);
    return Mem\_with\_type[address, size, accdesc];

// MemA[] - assignment form
// =====

MemA(bits(32) address, integer size) = bits(8*size) value
    boolean acqrel = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescExLDST(MemOp\_STORE, acqrel, tagchecked);
    Mem\_with\_type[address, size, accdesc] = value;
    return;
```

## Library pseudocode for aarch32/functions/memory/MemO

```
// MemO[] - non-assignment form
// =====

bits(8*size) MemO(bits(32) address, integer size)
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp\_LOAD, tagchecked);
    return Mem\_with\_type[address, size, accdesc];

// MemO[] - assignment form
// =====

MemO(bits(32) address, integer size) = bits(8*size) value
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp\_STORE, tagchecked);
    Mem\_with\_type[address, size, accdesc] = value;
    return;
```

## Library pseudocode for aarch32/functions/memory/MemS

```
// MemS[] - non-assignment form
// =====
// Memory accessor for streaming load multiple instructions

bits(8*size) MemS(bits(32) address, integer size)
    AccessDescriptor accdesc = CreateAccDescA32LSMD(MemOp\_LOAD);
    return Mem\_with\_type[address, size, accdesc];

// MemS[] - assignment form
// =====
// Memory accessor for streaming store multiple instructions

MemS(bits(32) address, integer size) = bits(8*size) value
    AccessDescriptor accdesc = CreateAccDescA32LSMD(MemOp\_STORE);
    Mem\_with\_type[address, size, accdesc] = value;
    return;
```

## Library pseudocode for aarch32/functions/memory/MemU

```
// MemU[] - non-assignment form
// =====

bits(8*size) MemU(bits(32) address, integer size)
    boolean nontemporal = FALSE;
    boolean privileged = PSTATE.EL != EL0;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_LOAD, nontemporal, privileged, tagchecked);
    return Mem\_with\_type[address, size, accdesc];

// MemU[] - assignment form
// =====

MemU(bits(32) address, integer size) = bits(8*size) value
    boolean nontemporal = FALSE;
    boolean privileged = PSTATE.EL != EL0;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_STORE, nontemporal, privileged, tagchecked);
    Mem\_with\_type[address, size, accdesc] = value;
    return;
```

## Library pseudocode for aarch32/functions/memory/MemU\_unpriv

```
// MemU_unpriv[] - non-assignment form
// =====

bits(8*size) MemU_unpriv[bits(32) address, integer size]
    boolean nontemporal = FALSE;
    boolean privileged = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_LOAD, nontemporal, privileged, tagchecked);
    return Mem\_with\_type[address, size, accdesc];

// MemU_unpriv[] - assignment form
// =====

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
    boolean nontemporal = FALSE;
    boolean privileged = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_STORE, nontemporal, privileged, tagchecked);
    Mem\_with\_type[address, size, accdesc] = value;
    return;
```



```

// Mem_with_type[] - non-assignment (read) form
// =====
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch32.MemSingle directly.

bits(size*8) Mem_with_type[bits(32) address, integer size, AccessDescriptor accdesc_in]
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    bits(size * 8) value;
    AccessDescriptor accdesc = accdesc_in;

    // Check alignment on size of element accessed, not overall access size
    integer alignment = if accdesc.ispair then halfsize else size;
    boolean aligned    = IsAligned(address, alignment);

    if !aligned && AArch32.UnalignedAccessFaults(accdesc) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    if aligned then
        value = AArch32.MemSingle[address, size, accdesc, aligned];
    else
        assert size > 1;
        value<7:0> = AArch32.MemSingle[address, 1, accdesc, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable(Unpredictable\_DEVPAGE2);
        assert c IN {Constraint\_FAULT, Constraint\_NONE};
        if c == Constraint\_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, accdesc, aligned];

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);

    return value;

// Mem_with_type[] - assignment (write) form
// =====
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem_with_type[bits(32) address, integer size, AccessDescriptor accdesc_in] = bits(size*8) value_in
    constant halfsize = size DIV 2;
    bits(size*8) value = value_in;
    AccessDescriptor accdesc = accdesc_in;

    // Check alignment on size of element accessed, not overall access size
    integer alignment = if accdesc.ispair then halfsize else size;
    boolean aligned    = IsAligned(address, alignment);

    if !aligned && AArch32.UnalignedAccessFaults(accdesc) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);

    if aligned then
        AArch32.MemSingle[address, size, accdesc, aligned] = value;
    else
        assert size > 1;
        AArch32.MemSingle[address, 1, accdesc, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.

        c = ConstrainUnpredictable(Unpredictable\_DEVPAGE2);
        assert c IN {Constraint\_FAULT, Constraint\_NONE};

```



```

    if c == Constraint\_NONE then aligned = TRUE;

    for i = 1 to size-1
        AArch32.MemSingle[address+i, 1, accdesc, aligned] = value<8*i+7:8*i>;
    return;

```

## Library pseudocode for aarch32/functions/ras/AArch32.ESBOperation

```

// AArch32.ESBOperation()
// =====
// Perform the AArch32 ESB operation for ESB executed in AArch32 state

AArch32.ESBOperation()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1';
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = EffectiveEA() == '1';

    if route_to_aarch64 then
        AArch64.ESBOperation();
        return;

    route_to_monitor = HaveEL(EL3) && ELUsingAArch32(EL3) && EffectiveEA() == '1';
    route_to_hyp = PSTATE.EL IN {EL0, EL1} && EL2Enabled() && (HCR.TGE == '1' || HCR.AMO == '1');

    bits(5) target;
    if route_to_monitor then
        target = M32\_Monitor;
    elsif route_to_hyp || PSTATE.M == M32\_Hyp then
        target = M32\_Hyp;
    else
        target = M32\_Abort;

    boolean mask_active;
    if CurrentSecurityState() == SS\_Secure then
        mask_active = TRUE;
    elsif target == M32\_Monitor then
        mask_active = SCR.AW == '1' && (!HaveEL(EL2) || (HCR.TGE == '0' && HCR.AMO == '0'));
    else
        mask_active = target == M32\_Abort || PSTATE.M == M32\_Hyp;

    mask_set = PSTATE.A == '1';
    (-, el) = ELFromM32(target);
    intdis = Halted() || ExternalDebugInterruptsDisabled(el);
    masked = intdis || (mask_active && mask_set);

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        bits(32) syndrome = Zeros(32);
        syndrome<31> = '1'; // A
        syndrome<15:0> = AArch32.PhysicalSErrorSyndrome();
        DISR = syndrome;
        ClearPendingPhysicalSError();

    return;

```

## Library pseudocode for aarch32/functions/ras/AArch32.EncodeAsyncErrorSyndrome

```
// AArch32.EncodeAsyncErrorSyndrome()
// =====
// Return the corresponding encoding for ErrorState.

bits(2) AArch32.EncodeAsyncErrorSyndrome(ErrorState errorstate)
    case errorstate of
        when ErrorState\_UC    return '00';
        when ErrorState\_UEU   return '01';
        when ErrorState\_UEO   return '10';
        when ErrorState\_UER   return '11';
        otherwise Unreachable();
```

## Library pseudocode for aarch32/functions/ras/AArch32.PhysicalSErrorSyndrome

```
// AArch32.PhysicalSErrorSyndrome()
// =====
// Generate SError syndrome.

bits(16) AArch32.PhysicalSErrorSyndrome()
    bits(32) syndrome = Zeros(32);
    FaultRecord fault = GetPendingPhysicalError();
    if PSTATE.EL == EL2 then
        ErrorState errstate = AArch32.PEErrorState(fault);
        syndrome<11:10> = AArch32.EncodeAsyncErrorSyndrome(errstate);    // AET
        syndrome<9>      = fault.extflag;                               // EA
        syndrome<5:0>    = '010001';                                   // DFSC
    else
        boolean long_format = TTBCR.EAE == '1';
        syndrome = AArch32.CommonFaultStatus(fault, long_format);
    return syndrome<15:0>;
```

## Library pseudocode for aarch32/functions/ras/AArch32.vESBOperation

```
// AArch32.vESBOperation()
// =====
// Perform the ESB operation for virtual SError interrupts executed in AArch32 state

AArch32.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // Check for EL2 using AArch64 state
    if !ELUsingAArch32(EL2) then
        AArch64.vESBOperation();
        return;

    // If physical SError interrupts are routed to Hyp mode, and TGE is not set,
    // then a virtual SError interrupt might be pending
    vSEI_enabled = HCR.TGE == '0' && HCR.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR.VA == '1';
    vintdis      = Halted() || ExternalDebugInterruptsDisabled(EL1);
    vmasked      = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        bits(32) syndrome = Zeros(32);
        syndrome<31> = '1';           // A
        syndrome<15:14> = VDFSR<15:14>; // AET
        syndrome<12> = VDFSR<12>;      // ExT
        syndrome<9> = TTBCR.EAE;       // LPAE
        if TTBCR.EAE == '1' then       // Long-descriptor format
            syndrome<5:0> = '010001'; // STATUS
        else                           // Short-descriptor format
            syndrome<10,3:0> = '10110'; // FS
        VDISR = syndrome;
        HCR.VA = '0';                 // Clear pending virtual SError

    return;
```

## Library pseudocode for aarch32/functions/registers/AArch32.ResetGeneralRegisters

```
// AArch32.ResetGeneralRegisters()
// =====

AArch32.ResetGeneralRegisters()

    for i = 0 to 7
        R[i] = bits(32) UNKNOWN;
    for i = 8 to 12
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
    if HaveEL(EL2) then Rmode[13, M32_Hyp] = bits(32) UNKNOWN; // No R14_hyp
    for i = 13 to 14
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
        Rmode[i, M32_IRQ] = bits(32) UNKNOWN;
        Rmode[i, M32_Svc] = bits(32) UNKNOWN;
        Rmode[i, M32_Abort] = bits(32) UNKNOWN;
        Rmode[i, M32_Undef] = bits(32) UNKNOWN;
        if HaveEL(EL3) then Rmode[i, M32_Monitor] = bits(32) UNKNOWN;

    return;
```

### Library pseudocode for aarch32/functions/registers/AArch32.ResetSIMDFPRegisters

```
// AArch32.ResetSIMDFPRegisters()
// =====

AArch32.ResetSIMDFPRegisters()

    for i = 0 to 15
        Q[i] = bits(128) UNKNOWN;

    return;
```

### Library pseudocode for aarch32/functions/registers/AArch32.ResetSpecialRegisters

```
// AArch32.ResetSpecialRegisters()
// =====

AArch32.ResetSpecialRegisters()

    // AArch32 special registers
    SPSR_fiq<31:0> = bits(32) UNKNOWN;
    SPSR_irq<31:0> = bits(32) UNKNOWN;
    SPSR_svc<31:0> = bits(32) UNKNOWN;
    SPSR_abt<31:0> = bits(32) UNKNOWN;
    SPSR_und<31:0> = bits(32) UNKNOWN;
    if HaveEL(EL2) then
        SPSR_hyp = bits(32) UNKNOWN;
        ELR_hyp = bits(32) UNKNOWN;
    if HaveEL(EL3) then
        SPSR_mon = bits(32) UNKNOWN;

    // External debug special registers
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;

    return;
```

### Library pseudocode for aarch32/functions/registers/AArch32.ResetSystemRegisters

```
// AArch32.ResetSystemRegisters()
// =====

AArch32.ResetSystemRegisters(boolean cold_reset);
```

### Library pseudocode for aarch32/functions/registers/ALUEXceptionReturn

```
// ALUEXceptionReturn()
// =====

ALUEXceptionReturn(bits(32) address)
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32 User, M32 System} then
        Constraint c = ConstrainUnpredictable(Unpredictable ALUEXCEPTIONRETURN);
        assert c IN {Constraint UNDEF, Constraint NOP};
        case c of
            when Constraint UNDEF
                UNDEFINED;
            when Constraint NOP
                EndOfInstruction();
        else
            AArch32.ExceptionReturn(address, SPSR\_curr[]);
```

## Library pseudocode for aarch32/functions/registers/ALUWritePC

```
// ALUWritePC()
// =====

ALUWritePC(bits(32) address)
  if CurrentInstrSet() == InstrSet\_A32 then
    BXWritePC(address, BranchType\_INDIR);
  else
    BranchWritePC(address, BranchType\_INDIR);
```

## Library pseudocode for aarch32/functions/registers/BXWritePC

```
// BXWritePC()
// =====

BXWritePC(bits(32) address_in, BranchType branch_type)
  bits(32) address = address_in;
  if address<0> == '1' then
    SelectInstrSet(InstrSet\_T32);
    address<0> = '0';
  else
    SelectInstrSet(InstrSet\_A32);
    // For branches to an unaligned PC counter in A32 state, the processor takes the branch
    // and does one of:
    // * Forces the address to be aligned
    // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
    if address<1> == '1' && ConstrainUnpredictableBool(Unpredictable\_A32FORCEALIGNPC) then
      address<1> = '0';
  boolean branch_conditional = !(AArch32.CurrentCond() IN {'111x'});
  BranchTo(address, branch_type, branch_conditional);
```

## Library pseudocode for aarch32/functions/registers/BranchWritePC

```
// BranchWritePC()
// =====

BranchWritePC(bits(32) address_in, BranchType branch_type)
  bits(32) address = address_in;
  if CurrentInstrSet() == InstrSet\_A32 then
    address<1:0> = '00';
  else
    address<0> = '0';
  boolean branch_conditional = !(AArch32.CurrentCond() IN {'111x'});
  BranchTo(address, branch_type, branch_conditional);
```

## Library pseudocode for aarch32/functions/registers/CBWritePC

```
// CBWritePC()
// =====
// Takes a branch from a CBNZ/CBZ instruction.

CBWritePC(bits(32) address_in)
  bits(32) address = address_in;
  assert CurrentInstrSet() == InstrSet\_T32;
  address<0> = '0';
  boolean branch_conditional = TRUE;
  BranchTo(address, BranchType\_DIR, branch_conditional);
```

### Library pseudocode for aarch32/functions/registers/D

```
// D[] - non-assignment form
// =====

bits(64) D[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    bits(128) vreg = V[n DIV 2, 128];
    return vreg<base+63:base>;

// D[] - assignment form
// =====

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    bits(128) vreg = V[n DIV 2, 128];
    vreg<base+63:base> = value;
    V[n DIV 2, 128] = vreg;
    return;
```

### Library pseudocode for aarch32/functions/registers/Din

```
// Din[] - non-assignment form
// =====

bits(64) Din[integer n]
    assert n >= 0 && n <= 31;
    return _Dclone[n];
```

### Library pseudocode for aarch32/functions/registers/LR

```
// LR - assignment form
// =====

LR = bits(32) value
    R[14] = value;
    return;

// LR - non-assignment form
// =====

bits(32) LR
    return R[14];
```

### Library pseudocode for aarch32/functions/registers/LoadWritePC

```
// LoadWritePC()
// =====

LoadWritePC(bits(32) address)
    BXWritePC(address, BranchType\_INDIR);
```

## Library pseudocode for aarch32/functions/registers/LookUpRIndex

```
// LookUpRIndex()
// =====

integer LookUpRIndex(integer n, bits(5) mode)
    assert n >= 0 && n <= 14;

    integer result;
    case n of // Select index by mode:      usr fiq irq svc abt und hyp
        when 8      result = RBankSelect(mode, 8, 24, 8, 8, 8, 8, 8);
        when 9      result = RBankSelect(mode, 9, 25, 9, 9, 9, 9, 9);
        when 10     result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
        when 11     result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
        when 12     result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
        when 13     result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
        when 14     result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
        otherwise   result = n;

    return result;
```

## Library pseudocode for aarch32/functions/registers/Monitor\_mode\_registers

```
bits(32) SP_mon;
bits(32) LR_mon;
```

## Library pseudocode for aarch32/functions/registers/PC

```
// PC - non-assignment form
// =====

bits(32) PC
    return R[15]; // This includes the offset from AArch32 state
```

## Library pseudocode for aarch32/functions/registers/PCStoreValue

```
// PCStoreValue()
// =====

bits(32) PCStoreValue()
    // This function returns the PC value. On architecture versions before Armv7, it
    // is permitted to instead return PC+4, provided it does so consistently. It is
    // used only to describe A32 instructions, so it returns the address of the current
    // instruction plus 8 (normally) or 12 (when the alternative is permitted).
    return PC;
```

## Library pseudocode for aarch32/functions/registers/Q

```
// Q[] - non-assignment form
// =====

bits(128) Q[integer n]
    assert n >= 0 && n <= 15;
    return V[n, 128];

// Q[] - assignment form
// =====

Q[integer n] = bits(128) value
    assert n >= 0 && n <= 15;
    V[n, 128] = value;
    return;
```

## Library pseudocode for aarch32/functions/registers/Qin

```
// Qin[] - non-assignment form
// =====

bits(128) Qin[integer n]
    assert n >= 0 && n <= 15;
    return Din[2*n+1]:Din[2*n];
```

## Library pseudocode for aarch32/functions/registers/R

```
// R[] - assignment form
// =====

R[integer n] = bits(32) value
    Rmode[n, PSTATE.M] = value;
    return;

// R[] - non-assignment form
// =====

bits(32) R[integer n]
    if n == 15 then
        offset = (if CurrentInstrSet() == InstrSet\_A32 then 8 else 4);
        return _PC<31:0> + offset;
    else
        return Rmode[n, PSTATE.M];
```

## Library pseudocode for aarch32/functions/registers/RBankSelect

```
// RBankSelect()
// =====

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq,
                    integer svc, integer abt, integer und, integer hyp)

    integer result;
    case mode of
        when M32\_User      result = usr; // User mode
        when M32\_FIQ      result = fiq; // FIQ mode
        when M32\_IRQ      result = irq; // IRQ mode
        when M32\_Svc      result = svc; // Supervisor mode
        when M32\_Abort     result = abt; // Abort mode
        when M32\_Hyp      result = hyp; // Hyp mode
        when M32\_Undef     result = und; // Undefined mode
        when M32\_System   result = usr; // System mode uses User mode registers
        otherwise         Unreachable(); // Monitor mode

    return result;
```



## Library pseudocode for aarch32/functions/registers/Rmode

```
// Rmode[] - non-assignment form
// =====

bits(32) Rmode[integer n, bits(5) mode]
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if CurrentSecurityState() != SS\_Secure then assert mode != M32\_Monitor;
    assert !BadMode(mode);

    if mode == M32\_Monitor then
        if n == 13 then return SP_mon;
        elsif n == 14 then return LR_mon;
        else return _R[n]<31:0>;
    else
        return _R[LookUpRIndex(n, mode)]<31:0>;

// Rmode[] - assignment form
// =====

Rmode[integer n, bits(5) mode] = bits(32) value
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if CurrentSecurityState() != SS\_Secure then assert mode != M32\_Monitor;
    assert !BadMode(mode);

    if mode == M32\_Monitor then
        if n == 13 then SP_mon = value;
        elsif n == 14 then LR_mon = value;
        else _R[n]<31:0> = value;
    else
        // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the X
        // register are unchanged or set to zero. This is also tested for on
        // exception entry, as this applies to all AArch32 registers.
        if HaveAArch64() && ConstrainUnpredictableBool(Unpredictable\_ZEROUPPER) then
            _R[LookUpRIndex(n, mode)] = ZeroExtend(value, 64);
        else
            _R[LookUpRIndex(n, mode)]<31:0> = value;

    return;
```

## Library pseudocode for aarch32/functions/registers/S

```
// S[] - non-assignment form
// =====

bits(32) S[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    bits(128) vreg = V[n DIV 4, 128];
    return vreg<base+31:base>;

// S[] - assignment form
// =====

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    bits(128) vreg = V[n DIV 4, 128];
    vreg<base+31:base> = value;
    V[n DIV 4, 128] = vreg;
    return;
```

## Library pseudocode for aarch32/functions/registers/\_Dclone

```
// _Dclone[]
// =====
// Clone the 64-bit Advanced SIMD and VFP extension register bank for use as input to
// instruction pseudocode, to avoid read-after-write for Advanced SIMD and VFP operations.

array bits(64) _Dclone[0..31];
```

## Library pseudocode for aarch32/functions/system/AArch32.ExceptionReturn

```
// AArch32.ExceptionReturn()
// =====

AArch32.ExceptionReturn(bits(32) new_pc_in, bits(32) spsr)
    bits(32) new_pc = new_pc_in;
    SynchronizeContext();
    // Attempts to change to an illegal mode or state will invoke the Illegal Execution state
    // mechanism
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' then
        // If the exception return is illegal, PC[1:0] are UNKNOWN
        new_pc<1:0> = bits(2) UNKNOWN;
    else
        // LR[1:0] or LR[0] are treated as being 0, depending on the target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0'; // T32
        else
            new_pc<1:0> = '00'; // A32

    boolean branch_conditional = !(AArch32.CurrentCond() IN {'111x'});
    BranchTo(new_pc, BranchType ERET, branch_conditional);

    CheckExceptionCatch(FALSE); // Check for debug event on exception return
```

## Library pseudocode for aarch32/functions/system/AArch32.ExecutingCP10or11Instr

```
// AArch32.ExecutingCP10or11Instr()
// =====

boolean AArch32.ExecutingCP10or11Instr()
    instr = ThisInstr();
    instr_set = CurrentInstrSet();
    assert instr_set IN {InstrSet\_A32, InstrSet\_T32};

    if instr_set == InstrSet\_A32 then
        return ((instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> IN {'101x'});
    else // InstrSet_T32
        return (instr<31:28> IN {'111x'} && (instr<27:24> == '1110' || instr<27:25> == '110') &&
            instr<11:8> IN {'101x'});
```

## Library pseudocode for aarch32/functions/system/AArch32.ITAdvance

```
// AArch32.ITAdvance()
// =====

AArch32.ITAdvance()
    if PSTATE.IT<2:0> == '000' then
        PSTATE.IT = '00000000';
    else
        PSTATE.IT<4:0> = LSL(PSTATE.IT<4:0>, 1);
    return;
```

### Library pseudocode for aarch32/functions/system/AArch32.SysRegRead

```
// AArch32.SysRegRead()
// =====
// Read from a 32-bit AArch32 System register and write the register's contents to R[t].

AArch32.SysRegRead(integer cp_num, bits(32) instr, integer t);
```

### Library pseudocode for aarch32/functions/system/AArch32.SysRegRead64

```
// AArch32.SysRegRead64()
// =====
// Read from a 64-bit AArch32 System register and write the register's contents to R[t] and R[t2].

AArch32.SysRegRead64(integer cp_num, bits(32) instr, integer t, integer t2);
```

### Library pseudocode for aarch32/functions/system/AArch32.SysRegReadCanWriteAPSR

```
// AArch32.SysRegReadCanWriteAPSR()
// =====
// Determines whether the AArch32 System register read instruction can write to APSR flags.

boolean AArch32.SysRegReadCanWriteAPSR(integer cp_num, bits(32) instr)
    assert UsingAArch32();
    assert (cp_num IN {14,15});
    assert cp_num == UInt(instr<11:8>);

    opc1 = UInt(instr<23:21>);
    opc2 = UInt(instr<7:5>);
    CRn  = UInt(instr<19:16>);
    CRm  = UInt(instr<3:0>);

    if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0 then // DBGDSCRint
        return TRUE;

    return FALSE;
```

### Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite

```
// AArch32.SysRegWrite()
// =====
// Read the contents of R[t] and write to a 32-bit AArch32 System register.

AArch32.SysRegWrite(integer cp_num, bits(32) instr, integer t);
```

### Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite64

```
// AArch32.SysRegWrite64()
// =====
// Read the contents of R[t] and R[t2] and write to a 64-bit AArch32 System register.

AArch32.SysRegWrite64(integer cp_num, bits(32) instr, integer t, integer t2);
```

### Library pseudocode for aarch32/functions/system/AArch32.SysRegWriteM

```
// AArch32.SysRegWriteM()
// =====
// Read a value from a virtual address and write it to an AArch32 System register.

AArch32.SysRegWriteM(integer cp_num, bits(32) instr, bits(32) address);
```

## Library pseudocode for aarch32/functions/system/AArch32.WriteMode

```
// AArch32.WriteMode()
// =====
// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.

AArch32.WriteMode(bits(5) mode)
    (valid,el) = ELFromM32(mode);
    assert valid;
    PSTATE.M    = mode;
    PSTATE.EL   = el;
    PSTATE.nRW  = '1';
    PSTATE.SP   = (if mode IN {M32\_User,M32\_System} then '0' else '1');
    return;
```

## Library pseudocode for aarch32/functions/system/AArch32.WriteModeByInstr

```
// AArch32.WriteModeByInstr()
// =====
// Function for dealing with writes to PSTATE.M from an AArch32 instruction, and ensuring that
// illegal state changes are correctly flagged in PSTATE.IL.

AArch32.WriteModeByInstr(bits(5) mode)
    (valid,el) = ELFromM32(mode);

    // 'valid' is set to FALSE if 'mode' is invalid for this implementation or the current value
    // of SCR.NS/SCR_EL3.NS. Additionally, it is illegal for an instruction to write 'mode' to
    // PSTATE.EL if it would result in any of:
    // * A change to a mode that would cause entry to a higher Exception level.
    if UInt(el) > UInt(PSTATE.EL) then
        valid = FALSE;

    // * A change to or from Hyp mode.
    if (PSTATE.M == M32\_Hyp || mode == M32\_Hyp) && PSTATE.M != mode then
        valid = FALSE;

    // * When EL2 is implemented, the value of HCR.TGE is '1', a change to a Non-secure EL1 mode.
    if PSTATE.M == M32\_Monitor && HaveEL(EL2) && el == EL1 && SCR.NS == '1' && HCR.TGE == '1' then
        valid = FALSE;

    if !valid then
        PSTATE.IL = '1';
    else
        AArch32.WriteMode(mode);
```

## Library pseudocode for aarch32/functions/system/BadMode

```
// BadMode()
// =====

boolean BadMode(bits(5) mode)
// Return TRUE if 'mode' encodes a mode that is not valid for this implementation
boolean valid;
case mode of
    when M32\_Monitor
        valid = HaveAArch32EL\(EL3\);
    when M32\_Hyp
        valid = HaveAArch32EL\(EL2\);
    when M32\_FIQ, M32\_IRQ, M32\_Svc, M32\_Abort, M32\_Undef, M32\_System
        // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
        // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
        // AArch64, then these modes are EL1 modes.
        // Therefore it is sufficient to test this implementation supports EL1 using AArch32.
        valid = HaveAArch32EL\(EL1\);
    when M32\_User
        valid = HaveAArch32EL\(EL0\);
    otherwise
        valid = FALSE; // Passed an illegal mode value
return !valid;
```

## Library pseudocode for aarch32/functions/system/BankedRegisterAccessValid

```
// BankedRegisterAccessValid()
// =====
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.

BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)

case SYSm of
    when '000xx', '00100' // R8_usr to R12_usr
        if mode != M32\_FIQ then UNPREDICTABLE;
    when '00101' // SP_usr
        if mode == M32\_System then UNPREDICTABLE;
    when '00110' // LR_usr
        if mode IN {M32\_Hyp, M32\_System} then UNPREDICTABLE;
    when '010xx', '0110x', '01110' // R8_fiq to R12_fiq, SP_fiq, LR_fiq
        if mode == M32\_FIQ then UNPREDICTABLE;
    when '1000x' // LR_irq, SP_irq
        if mode == M32\_IRQ then UNPREDICTABLE;
    when '1001x' // LR_svc, SP_svc
        if mode == M32\_Svc then UNPREDICTABLE;
    when '1010x' // LR_abt, SP_abt
        if mode == M32\_Abort then UNPREDICTABLE;
    when '1011x' // LR_und, SP_und
        if mode == M32\_Undef then UNPREDICTABLE;
    when '1110x' // LR_mon, SP_mon
        if (!HaveEL\(EL3\) || CurrentSecurityState\(\) != SS\_Secure ||
            mode == M32\_Monitor) then UNPREDICTABLE;
    when '11110' // ELR_hyp, only from Monitor or Hyp mode
        if !HaveEL\(EL2\) || !(mode IN {M32\_Monitor, M32\_Hyp}) then UNPREDICTABLE;
    when '11111' // SP_hyp, only from Monitor mode
        if !HaveEL\(EL2\) || mode != M32\_Monitor then UNPREDICTABLE;
    otherwise
        UNPREDICTABLE;

return;
```

## Library pseudocode for aarch32/functions/system/CPSRWriteByInstr

```
// CPSRWriteByInstr()
// =====
// Update PSTATE.<N,Z,C,V,Q,GE,E,A,I,F,M> from a CPSR value written by an MSR instruction.

CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    privileged = PSTATE.EL != EL0; // PSTATE.<A,I,F,M> are not writable at EL0

    // Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
    if bytemask<3> == '1' then
        PSTATE.<N,Z,C,V,Q> = value<31:27>;
        // Bits <26:24> are ignored

    if bytemask<2> == '1' then
        if HaveSSBSExt() then
            PSTATE.SSBS = value<23>;
        if privileged then
            PSTATE.PAN = value<22>;
        if HaveDITExt() then
            PSTATE.DIT = value<21>;
        // Bit <20> is RES0
        PSTATE.GE = value<19:16>;

    if bytemask<1> == '1' then
        // Bits <15:10> are RES0
        PSTATE.E = value<9>; // PSTATE.E is writable at EL0
        if privileged then
            PSTATE.A = value<8>;

    if bytemask<0> == '1' then
        if privileged then
            PSTATE.<I,F> = value<7:6>;
            // Bit <5> is RES0
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32.WriteModeByInstr(value<4:0>);

    return;
```

## Library pseudocode for aarch32/functions/system/ConditionPassed

```
// ConditionPassed()
// =====

boolean ConditionPassed()
    return ConditionHolds(AArch32.CurrentCond());
```

## Library pseudocode for aarch32/functions/system/CurrentCond

```
// CurrentCond()
// =====

bits(4) AArch32.CurrentCond();
```

## Library pseudocode for aarch32/functions/system/InITBlock

```
// InITBlock()
// =====

boolean InITBlock()
    if CurrentInstrSet() == InstrSet\_T32 then
        return PSTATE.IT<3:0> != '0000';
    else
        return FALSE;
```

## Library pseudocode for aarch32/functions/system/LastInITBlock

```
// LastInITBlock()
// =====

boolean LastInITBlock()
    return (PSTATE.IT<3:0> == '1000');
```

## Library pseudocode for aarch32/functions/system/SPSRWriteByInstr

```
// SPSRWriteByInstr()
// =====

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)

    bits(32) new_spsr = SPSR\_curr[];

    if bytemask<3> == '1' then
        new_spsr<31:24> = value<31:24>; // N,Z,C,V,Q flags, IT[1:0],J bits

    if bytemask<2> == '1' then
        new_spsr<23:16> = value<23:16>; // IL bit, GE[3:0] flags

    if bytemask<1> == '1' then
        new_spsr<15:8> = value<15:8>; // IT[7:2] bits, E bit, A interrupt mask

    if bytemask<0> == '1' then
        new_spsr<7:0> = value<7:0>; // I,F interrupt masks, T bit, Mode bits

    SPSR\_curr[] = new_spsr; // UNPREDICTABLE if User or System mode

    return;
```

## Library pseudocode for aarch32/functions/system/SPSRAccessValid

```
// SPSRAccessValid()
// =====
// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

SPSRAccessValid(bits(5) SYSm, bits(5) mode)
    case SYSm of
        when '01110' // SPSR_fiq
            if mode == M32\_FIQ then UNPREDICTABLE;
        when '10000' // SPSR_irq
            if mode == M32\_IRQ then UNPREDICTABLE;
        when '10010' // SPSR_svc
            if mode == M32\_Svc then UNPREDICTABLE;
        when '10100' // SPSR_abt
            if mode == M32\_Abort then UNPREDICTABLE;
        when '10110' // SPSR_und
            if mode == M32\_Undef then UNPREDICTABLE;
        when '11100' // SPSR_mon
            if (!HaveEL(EL3) || mode == M32\_Monitor ||
                CurrentSecurityState() != SS\_Secure) then UNPREDICTABLE;
        when '11110' // SPSR_hyp
            if !HaveEL(EL2) || mode != M32\_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;

    return;
```

## Library pseudocode for aarch32/functions/system/SelectInstrSet

```
// SelectInstrSet()
// =====

SelectInstrSet(InstrSet iset)
    assert CurrentInstrSet() IN {InstrSet\_A32, InstrSet\_T32};
    assert iset IN {InstrSet\_A32, InstrSet\_T32};

    PSTATE.T = if iset == InstrSet\_A32 then '0' else '1';

    return;
```

## Library pseudocode for aarch32/functions/tlbi/AArch32.DTLBI\_ALL

```
// AArch32.DTLBI_ALL()
// =====
// Invalidate all data TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

AArch32.DTLBI_ALL(SecurityState security, Regime regime, Shareability shareability,
    TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op          = TLBIOp\_DALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel\_Any;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch32/functions/tlbi/AArch32.DTLBI\_ASID

```
// AArch32.DTLBI_ASID()
// =====
// Invalidate all data TLB stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.DTLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op          = TLBIOp\_DASID;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = TLBILevel\_Any;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```



## Library pseudocode for aarch32/functions/tlbi/AArch32.DTLBI\_VA

```
// AArch32.DTLBI_VA()
// =====
// Invalidate by VA all stage 1 data TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.DTLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_DVA;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid = Zeros(8) : Rt<7:0>;
    r.address = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch32/functions/tlbi/AArch32.ITLBI\_ALL

```
// AArch32.ITLBI_ALL()
// =====
// Invalidate all instruction TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

AArch32.ITLBI_ALL(SecurityState security, Regime regime, Shareability shareability,
                 TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_IALL;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.level = TLBILevel\_Any;
    r.attr = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch32/functions/tlbi/AArch32.ITLBI\_ASID

```
// AArch32.ITLBI_ASID()
// =====
// Invalidate all instruction TLB stage 1 entries matching the indicated VMID
// (where regime supports) and ASID in the parameter Rt in the indicated translation
// regime with the indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.ITLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
                   Shareability shareability, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op          = TLBIOp\_IASID;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = TLBILevel\_Any;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch32/functions/tlbi/AArch32.ITLBI\_VA

```
// AArch32.ITLBI_VA()
// =====
// Invalidate by VA all stage 1 instruction TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.ITLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op          = TLBIOp\_IVA;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;
    r.address      = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI\_ALL

```
// AArch32.TLBI_ALL()
// =====
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

AArch32.TLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op = TLBIOp\_ALL;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.level = TLBILevel\_Any;
    r.attr = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI\_ASID

```
// AArch32.TLBI_ASID()
// =====
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_ASID;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = TLBILevel\_Any;
    r.attr = attr;
    r.asid = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI\_IPAS2

```
// AArch32.TLBI_IPAS2()
// =====
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Rt.

AArch32.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                   Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2};
    assert security == SS\_NonSecure;

    TLBIRecord r;
    r.op = TLBIOp\_IPAS2;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.address = Zeros(24) : Rt<27:0> : Zeros(12);
    r.ipaspace = PAS\_NonSecure;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI\_VA

```
// AArch32.TLBI_VA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_VA;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid = Zeros(8) : Rt<7:0>;
    r.address = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI\_VAA

```
// AArch32.TLBI_VAA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_VAA;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.address = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI\_VMALL

```
// AArch32.TLBI_VMALL()
// =====
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.

AArch32.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_VMALL;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.level = TLBILevel\_Any;
    r.vmid = vmid;
    r.attr = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI\_VMALLS12

```
// AArch32.TLBI_VMALLS12()
// =====
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.

AArch32.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op = TLBIOp\_VMALLS12;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.level = TLBILevel\_Any;
    r.vmid = vmid;
    r.attr = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch32/functions/v6simd/Sat

```
// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;
```

## Library pseudocode for aarch32/functions/v6simd/SignedSat

```
// SignedSat()
// =====

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;
```

## Library pseudocode for aarch32/functions/v6simd/UnsignedSat

```
// UnsignedSat()
// =====

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
```

## Library pseudocode for aarch32/ic/AArch32.IC

```
// AArch32.IC()
// =====
// Perform Instruction Cache Operation.

AArch32.IC(CacheOpScope opscope)
    regval = bits(32) UNKNOWN;
    AArch32.IC(regval, opscope);

// AArch32.IC()
// =====
// Perform Instruction Cache Operation.

AArch32.IC(bits(32) regval, CacheOpScope opscope)
    CacheRecord cache;

    cache.acctype = AccessType_IC;
    cache.cachetype = CacheType_Instruction;
    cache.cacheop = CacheOp_Invalidate;
    cache.opscope = opscope;
    cache.security = SecurityStateAtEL(PSTATE.EL);

    if opscope IN {CacheOpScope_ALLU, CacheOpScope_ALLUIS} then
        if opscope == CacheOpScope_ALLUIS || (opscope == CacheOpScope_ALLU && PSTATE.EL == EL1
            && EL2Enabled() && HCR.FB == '1') then
            cache.shareability = Shareability_ISH;
        else
            cache.shareability = Shareability_NSH;
        cache.regval = ZeroExtend(regval, 64);
        CACHE_OP(cache);
    else
        assert opscope == CacheOpScope_PoU;

        if EL2Enabled() then
            if PSTATE.EL IN {EL0, EL1} then
                cache.is_vmid_valid = TRUE;
                cache.vmid = VMID[];
            else
                cache.is_vmid_valid = FALSE;
        else
            cache.is_vmid_valid = FALSE;

        if PSTATE.EL == EL0 then
            cache.is_asid_valid = TRUE;
            cache.asid = ASID[];
        else
            cache.is_asid_valid = FALSE;

        need_translate = ICInstNeedsTranslation(opscope);

        cache.shareability = Shareability_NSH;
        cache.vaddress = ZeroExtend(regval, 64);
        cache.translated = need_translate;

        if !need_translate then
            cache.paddress = FullAddress UNKNOWN;
            CACHE_OP(cache);
            return;

        integer size = 0;
        boolean aligned = TRUE;
        AccessDescriptor accdesc = CreateAccDescIC(cache);
        AddressDescriptor memaddrdesc = AArch32.TranslateAddress(regval, accdesc, aligned, size);
        if IsFault(memaddrdesc) then
            AArch32.Abort(regval, memaddrdesc.fault);

        cache.paddress = memaddrdesc.paddress;
        CACHE_OP(cache);
    return;
```

## Library pseudocode for aarch32/predictionrestrict/AArch32.RestrictPrediction

```
// AArch32.RestrictPrediction()
// =====
// Clear all predictions in the context.

AArch32.RestrictPrediction(bits(32) val, RestrictType restriction)

    ExecutionCntxt c;
    target_el      = val<25:24>;

    // If the target EL is not implemented or the instruction is executed at an
    // EL lower than the specified level, the instruction is treated as a NOP.
    if !HaveEL(target_el) || UInt(target_el) > UInt(PSTATE.EL) then EndOfInstruction();

    bit ns  = val<26>;
    bit nse = bit UNKNOWN;
    ss = TargetSecurityState(ns, nse);

    c.security = ss;
    c.target_el = target_el;

    if EL2Enabled() then
        if PSTATE.EL IN {EL0, EL1} then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = FALSE;
            c.vmid          = VMID[];

            elseif target_el IN {EL0, EL1} then
                c.is_vmid_valid = TRUE;
                c.all_vmid      = val<27> == '1';
                c.vmid          = ZeroExtend(val<23:16>, 16);           // Only valid if val<27> == '0';
            else
                c.is_vmid_valid = FALSE;
        else
            c.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = FALSE;
        c.asid          = ASID[];

    elseif target_el == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = val<8> == '1';
        c.asid          = ZeroExtend(val<7:0>, 16);           // Only valid if val<8> == '0';

    else
        c.is_asid_valid = FALSE;

    c.restriction = restriction;
    RESTRICT_PREDICTIONS(c);
```





```

// AArch32.DefaultTEXDecode()
// =====
// Apply short-descriptor format memory region attributes, without TEX remap

MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX_in, bit C_in, bit B_in, bit s)
    MemoryAttributes memattrs;
    bits(3) TEX = TEX_in;
    bit C = C_in;
    bit B = B_in;

    // Reserved values map to allocated values
    if (TEX == '001' && C:B == '01') || (TEX == '010' && C:B != '00') || TEX == '011' then
        bits(5) texcb;
        (-, texcb) = ConstrainUnpredictableBits(Unpredictable\_RESTEXCB, 5);
        TEX = texcb<4:2>; C = texcb<1>; B = texcb<0>;

    // Distinction between Inner Shareable and Outer Shareable is not supported in this format
    // A memory region is either Non-shareable or Outer Shareable
    case TEX:C:B of
        when '00000'
            // Device-nGnRnE
            memattrs.memtype = MemType\_Device;
            memattrs.device = DeviceType\_nGnRnE;
            memattrs.shareability = Shareability\_OSH;
        when '00001', '01000'
            // Device-nGnRE
            memattrs.memtype = MemType\_Device;
            memattrs.device = DeviceType\_nGnRE;
            memattrs.shareability = Shareability\_OSH;
        when '00010'
            // Write-through Read allocate
            memattrs.memtype = MemType\_Normal;
            memattrs.inner.attrs = MemAttr\_WT;
            memattrs.inner.hints = MemHint\_RA;
            memattrs.outer.attrs = MemAttr\_WT;
            memattrs.outer.hints = MemHint\_RA;
            memattrs.shareability = if s == '1' then Shareability\_OSH else Shareability\_NSH;
        when '00011'
            // Write-back Read allocate
            memattrs.memtype = MemType\_Normal;
            memattrs.inner.attrs = MemAttr\_WB;
            memattrs.inner.hints = MemHint\_RA;
            memattrs.outer.attrs = MemAttr\_WB;
            memattrs.outer.hints = MemHint\_RA;
            memattrs.shareability = if s == '1' then Shareability\_OSH else Shareability\_NSH;
        when '00100'
            // Non-cacheable
            memattrs.memtype = MemType\_Normal;
            memattrs.inner.attrs = MemAttr\_NC;
            memattrs.outer.attrs = MemAttr\_NC;
            memattrs.shareability = Shareability\_OSH;
        when '00110'
            memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
        when '00111'
            // Write-back Read and Write allocate
            memattrs.memtype = MemType\_Normal;
            memattrs.inner.attrs = MemAttr\_WB;
            memattrs.inner.hints = MemHint\_RWA;
            memattrs.outer.attrs = MemAttr\_WB;
            memattrs.outer.hints = MemHint\_RWA;
            memattrs.shareability = if s == '1' then Shareability\_OSH else Shareability\_NSH;
        when '1xxxx'
            // Cacheable, TEX<1:0> = Outer attrs, {C,B} = Inner attrs
            memattrs.memtype = MemType\_Normal;
            memattrs.inner = DecodeSDFAttr(C:B);
            memattrs.outer = DecodeSDFAttr(TEX<1:0>);

            if memattrs.inner.attrs == MemAttr\_NC && memattrs.outer.attrs == MemAttr\_NC then
                memattrs.shareability = Shareability\_OSH;
            else

```

```

        memattrs.shareability = if s == '1' then Shareability\_OSH else Shareability\_NSH;
    otherwise
        // Reserved, handled above
        Unreachable();

    // The Transient hint is not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;
    memattrs.tags
        = MemTag\_Untagged;

    if memattrs.inner.attrs == MemAttr\_WB && memattrs.outer.attrs == MemAttr\_WB then
        memattrs.xs = '0';
    else
        memattrs.xs = '1';

    return memattrs;

```

### Library pseudocode for aarch32/translation/attrs/AArch32.MAIRAttr

```

// AArch32.MAIRAttr()
// =====
// Retrieve the memory attribute encoding indexed in the given MAIR

bits(8) AArch32.MAIRAttr(integer index, MAIRType mair)
    assert (index < 8);
    bit_index = 8 * index;
    return mair<bit_index+7:bit_index>;

```

## Library pseudocode for aarch32/translation/attrs/AArch32.RemappedTEXDecode

```
// AArch32.RemappedTEXDecode()
// =====
// Apply short-descriptor format memory region attributes, with TEX remap

MemoryAttributes AArch32.RemappedTEXDecode(Regime regime, bits(3) TEX, bit C, bit B, bit s)

MemoryAttributes memattrs;
PRRR_Type prrr;
NMRR_Type nmrr;

region = UInt(TEX<0>:C:B);          // TEX<2:1> are ignored in this mapping scheme
if region == 6 then
    return MemoryAttributes IMPLEMENTATION_DEFINED;

if regime == Regime\_EL30 then
    prrr = PRRR_S;
    nmrr = NMRR_S;
elseif HaveAArch32EL(EL3) then
    prrr = PRRR_NS;
    nmrr = NMRR_NS;
else
    prrr = PRRR;
    nmrr = NMRR;

base = 2 * region;
attrfield = prrr<base+1:base>;

if attrfield == '11' then          // Reserved, maps to allocated value
    (-, attrfield) = ConstrainUnpredictableBits(Unpredictable\_RESPPRR, 2);

case attrfield of
    when '00'                      // Device-nGnRnE
        memattrs.memtype           = MemType\_Device;
        memattrs.device            = DeviceType\_nGnRnE;
        memattrs.shareability      = Shareability\_OSH;
    when '01'                      // Device-nGnRE
        memattrs.memtype           = MemType\_Device;
        memattrs.device            = DeviceType\_nGnRE;
        memattrs.shareability      = Shareability\_OSH;
    when '10'
        NSn = if s == '0' then prrr.NS0 else prrr.NS1;
        NOSm = prrr<region+24> AND NSn;
        IRn = nmrr<base+1:base>;
        ORn = nmrr<base+17:base+16>;

        memattrs.memtype = MemType\_Normal;
        memattrs.inner   = DecodeSDFAttr(IRn);
        memattrs.outer   = DecodeSDFAttr(ORn);
        if memattrs.inner.attrs == MemAttr\_NC && memattrs.outer.attrs == MemAttr\_NC then
            memattrs.shareability = Shareability\_OSH;
        else
            bits(2) sh = NSn:NOSm;
            memattrs.shareability = DecodeShareability(sh);
    when '11'
        Unreachable();

// The Transient hint is not supported in this format
memattrs.inner.transient = FALSE;
memattrs.outer.transient = FALSE;
memattrs.tags            = MemTag\_Untagged;

if memattrs.inner.attrs == MemAttr\_WB && memattrs.outer.attrs == MemAttr\_WB then
    memattrs.xs = '0';
else
    memattrs.xs = '1';

return memattrs;
```

## Library pseudocode for aarch32/translation/debug/AArch32.CheckBreakpoint

```
// AArch32.CheckBreakpoint()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch32.CheckBreakpoint(FaultRecord fault_in, bits(32) vaddress,
                                     AccessDescriptor accdesc, integer size)

    assert ELUsingAArch32(S1TranslationRegime()) ;
    assert size IN {2,4};

    FaultRecord fault = fault_in;
    match = FALSE;
    mismatch = FALSE;

    for i = 0 to NumBreakpointsImplemented() - 1
        (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, accdesc, size);
        match = match || match_i;
        mismatch = mismatch || mismatch_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt\_Breakpoint;
        Halt(reason);
    elseif (match || mismatch) then
        fault.statuscode = Fault\_Debug;
        fault.debugmoe = DebugException\_Breakpoint;

    return fault;
```

## Library pseudocode for aarch32/translation/debug/AArch32.CheckDebug

```
// AArch32.CheckDebug()
// =====
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccessDescriptor accdesc, integer size)

    FaultRecord fault = NoFault(accdesc);

    boolean d_side = (IsDataAccess(accdesc.acctype) || accdesc.acctype == AccessType\_DC);
    boolean i_side = (accdesc.acctype == AccessType\_IFETCH);
    generate_exception = AArch32.GenerateDebugExceptions() && DBGDSCRext.MDBGGen == '1';
    halt = HaltOnBreakpointOrWatchpoint();
    // Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
    vector_catch_first = ConstrainUnpredictableBool(Unpredictable\_BPVECTORCATCHPRI);

    if i_side && vector_catch_first && generate_exception then
        fault = AArch32.CheckVectorCatch(fault, vaddress, size);

    if fault.statuscode == Fault\_None && (generate_exception || halt) then
        if d_side then
            fault = AArch32.CheckWatchpoint(fault, vaddress, accdesc, size);
        elseif i_side then
            fault = AArch32.CheckBreakpoint(fault, vaddress, accdesc, size);

    if fault.statuscode == Fault\_None && i_side && !vector_catch_first && generate_exception then
        return AArch32.CheckVectorCatch(fault, vaddress, size);

    return fault;
```

## Library pseudocode for aarch32/translation/debug/AArch32.CheckVectorCatch

```
// AArch32.CheckVectorCatch()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when debug exceptions are enabled.

FaultRecord AArch32.CheckVectorCatch(FaultRecord fault_in, bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    FaultRecord fault = fault_in;
    match = AArch32.VCRMATCH(vaddress);
    if size == 4 && !match && AArch32.VCRMATCH(vaddress + 2) then
        match = ConstrainUnpredictableBool(Unpredictable\_VCMATCHHALF);

    if match then
        fault.statuscode = Fault\_Debug;
        fault.debugmoe = DebugException\_VectorCatch;

    return fault;
```

## Library pseudocode for aarch32/translation/debug/AArch32.CheckWatchpoint

```
// AArch32.CheckWatchpoint()
// =====
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch32.CheckWatchpoint(FaultRecord fault_in, bits(32) vaddress,
                                     AccessDescriptor accdesc, integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    FaultRecord fault = fault_in;

    if accdesc.acctype == AccessType\_DC then
        if accdesc.cacheop != CacheOp\_Invalidate then
            return fault;
        elsif !(boolean IMPLEMENTATION_DEFINED "DCIMVAC generates watchpoint") then
            return fault;
    elsif IsDataAccess(accdesc.acctype) then
        return fault;

    match = FALSE;
    for i = 0 to NumWatchpointsImplemented() - 1
        if AArch32.WatchpointMatch(i, vaddress, size, accdesc) then
            match = TRUE;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt\_Watchpoint;
        EDWAR = ZeroExtend(vaddress, 64);
        Halt(reason);
    elsif match then
        fault.statuscode = Fault\_Debug;
        fault.debugmoe = DebugException\_Watchpoint;

    return fault;
```

## Library pseudocode for aarch32/translation/faults/AArch32.IPAIsOutOfRange

```
// AArch32.IPAIsOutOfRange()
// =====
// Check intermediate physical address bits not resolved by translation are ZERO

boolean AArch32.IPAIsOutOfRange(S2TWPParams walkparams, bits(40) ipa)
    // Input Address size
    iasize = AArch32.S2IASize(walkparams.t0sz);

    return iasize < 40 && IsZero(ipa<39:iasize>);
```

## Library pseudocode for aarch32/translation/faults/AArch32.S1HasAlignmentFault

```
// AArch32.S1HasAlignmentFault()
// =====
// Returns whether stage 1 output fails alignment requirement on data accesses
// to Device memory

boolean AArch32.S1HasAlignmentFault(AccessDescriptor accdesc, boolean aligned,
                                     bit ntlsmd, MemoryAttributes memattrs)
    if accdesc.acctype == AccessType\_IFETCH then
        return FALSE;
    elsif accdesc.a32lsmd && ntlsmd == '0' then
        return memattrs.memtype == MemType\_Device && memattrs.device != DeviceType\_GRE;
    elsif accdesc.acctype == AccessType\_DCZero then
        return memattrs.memtype == MemType\_Device;
    else
        return memattrs.memtype == MemType\_Device && !aligned;
```

## Library pseudocode for aarch32/translation/faults/AArch32.S1LDHasPermissionsFault

```
// AArch32.S1LDHasPermissionsFault()
// =====
// Returns whether an access using stage 1 long-descriptor translation
// violates permissions of target memory

boolean AArch32.S1LDHasPermissionsFault(Regime regime, S1TWPParams walkparams, Permissions perms,
                                         MemType memtype, PASpace paspace, AccessDescriptor accdesc)

    bit r, w, x;
    bit pr, pw;
    bit ur, uw;
    bit xn;
    if HasUnprivileged(regime) then
        // Apply leaf permissions
        case perms.ap<2:1> of
            when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
            when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL

        // Apply hierarchical permissions
        case perms.ap_table of
            when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
            when '01' (pr,pw,ur,uw) = ( pr, pw, '0','0'); // Privileged access
            when '10' (pr,pw,ur,uw) = ( pr, '0', ur, '0'); // Read-only
            when '11' (pr,pw,ur,uw) = ( pr, '0', '0','0'); // Read-only, privileged access

        xn = perms.xn OR perms.xn_table;
        pxn = perms.pxn OR perms.pxn_table;

        ux = ur AND NOT(xn OR (uw AND walkparams.wxn));
        px = pr AND NOT(xn OR pxn OR (pw AND walkparams.wxn) OR (uw AND walkparams.uwxn));

        if HavePANExt() && accdesc.pan then
            pan = PSTATE.PAN AND (ur OR uw);
            pr = pr AND NOT(pan);
            pw = pw AND NOT(pan);

        (r,w,x) = if accdesc.el == ELO then (ur,uw,ux) else (pr,pw,px);

        // Prevent execution from Non-secure space by PE in Secure state if SIF is set
        if accdesc.ss == SS Secure && paspace == PAS NonSecure then
            x = x AND NOT(walkparams.sif);
    else
        // Apply leaf permissions
        case perms.ap<2> of
            when '0' (r,w) = ('1','1'); // No effect
            when '1' (r,w) = ('1','0'); // Read-only

        // Apply hierarchical permissions
        case perms.ap_table<1> of
            when '0' (r,w) = ( r , w ); // No effect
            when '1' (r,w) = ( r , '0'); // Read-only

        xn = perms.xn OR perms.xn_table;
        x = NOT(xn OR (w AND walkparams.wxn));

    if accdesc.acctype == AccessType IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable INSTRDEVICE);
        if constraint == Constraint FAULT && memtype == MemType Device then
            return TRUE;
        else
            return x == '0';
    elsif accdesc.acctype IN {AccessType IC, AccessType DC} then
        return FALSE;
    elsif accdesc.write then
        return w == '0';
    else
        return r == '0';
```



## Library pseudocode for aarch32/translation/faults/AArch32.S1SDHasPermissionsFault

```
// AArch32.S1SDHasPermissionsFault()
// =====
// Returns whether an access using stage 1 short-descriptor translation
// violates permissions of target memory

boolean AArch32.S1SDHasPermissionsFault(Regime regime, Permissions perms_in, MemType memtype,
                                         PASpace paspace, AccessDescriptor accdesc)

    Permissions perms = perms_in;
    bit pr, pw;
    bit ur, uw;
    SCTLR_Type sctlr;
    if regime == Regime\_EL30 then
        sctlr = SCTLR_S;
    elsif HaveAArch32EL(EL3) then
        sctlr = SCTLR_NS;
    else
        sctlr = SCTLR;

    if sctlr.AFE == '0' then
        // Map Reserved encoding '100'
        if perms.ap == '100' then
            perms.ap = bits(3) IMPLEMENTATION_DEFINED "Reserved short descriptor AP encoding";

        case perms.ap of
            when '000' (pr,pw,ur,uw) = ('0','0','0','0'); // No access
            when '001' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '010' (pr,pw,ur,uw) = ('1','1','1','0'); // R/W at PL1, RO at PL0
            when '011' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            // '100' is reserved
            when '101' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
            when '110' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL (deprecated)
            when '111' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL
        else // Simplified access permissions model
            case perms.ap<2:1> of
                when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
                when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
                when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
                when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL

    ux = ur AND NOT(perms.xn OR (uw AND sctlr.WXN));
    px = pr AND NOT(perms.xn OR perms.pxn OR (pw AND sctlr.WXN) OR (uw AND sctlr.UWXN));

    if HavePANExt() && accdesc.pan then
        pan = PSTATE.PAN AND (ur OR uw);
        pr = pr AND NOT(pan);
        pw = pw AND NOT(pan);

    (r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

    // Prevent execution from Non-secure space by PE in Secure state if SIF is set
    if accdesc.ss == SS\_Secure && paspace == PAS\_NonSecure then
        x = x AND NOT(if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF);

    if accdesc.acctype == AccessType\_IFETCH then
        if (memtype == MemType\_Device &&
            ConstrainUnpredictable(Unpredictable\_INSTRDEVICE) == Constraint\_FAULT) then
            return TRUE;
        else
            return x == '0';
    elsif accdesc.acctype IN {AccessType\_IC, AccessType\_DC} then
        return FALSE;
    elsif accdesc.write then
        return w == '0';
    else
        return r == '0';
```

## Library pseudocode for aarch32/translation/faults/AArch32.S2HasAlignmentFault

```
// AArch32.S2HasAlignmentFault()
// =====
// Returns whether stage 2 output fails alignment requirement on data accesses
// to Device memory

boolean AArch32.S2HasAlignmentFault(AccessDescriptor accdesc, boolean aligned,
                                     MemoryAttributes memattrs)
    if accdesc.acctype == AccessType\_IFETCH then
        return FALSE;
    elsif accdesc.acctype == AccessType\_DCZero then
        return memattrs.memtype == MemType\_Device;
    else
        return memattrs.memtype == MemType\_Device && !aligned;
```

## Library pseudocode for aarch32/translation/faults/AArch32.S2HasPermissionsFault

```
// AArch32.S2HasPermissionsFault()
// =====
// Returns whether stage 2 access violates permissions of target memory

boolean AArch32.S2HasPermissionsFault(S2TTWParams walkparams, Permissions perms, MemType memtype,
                                       AccessDescriptor accdesc)
    bit px;
    bit ux;
    r = perms.s2ap<0>;
    w = perms.s2ap<1>;
    bit x;
    if HaveExtendedExecuteNeverExt() then
        case perms.s2xn:perms.s2xnx of
            when '00' (px, ux) = ( r , r );
            when '01' (px, ux) = ('0', r );
            when '10' (px, ux) = ('0', '0');
            when '11' (px, ux) = ( r , '0');

        x = if accdesc.el == EL0 then ux else px;
    else
        x = r AND NOT(perms.s2xn);

    if accdesc.acctype == AccessType\_TTW then
        return (walkparams.ptw == '1' && memtype == MemType\_Device) || r == '0';

    elsif accdesc.acctype == AccessType\_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable\_INSTRDEVICE);
        return (constraint == Constraint\_FAULT && memtype == MemType\_Device) || x == '0';

    elsif accdesc.acctype IN {AccessType\_IC, AccessType\_DC} then
        return FALSE;

    elsif accdesc.write then
        return w == '0';

    else
        return r == '0';
```

## Library pseudocode for aarch32/translation/faults/AArch32.S2InconsistentSL

```
// AArch32.S2InconsistentSL()
// =====
// Detect inconsistent configuration of stage 2 T0SZ and SL fields

boolean AArch32.S2InconsistentSL(S2TTWParams walkparams)
    startlevel = AArch32.S2StartLevel(walkparams.sl0);
    levels     = FINAL\_LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride     = granulebits - 3;

    // Input address size must at least be large enough to be resolved from the start level
    sl_min_iasize = (
        levels * stride // Bits resolved by table walk, except initial level
        + granulebits   // Bits directly mapped to output address
        + 1);           // At least 1 more bit to be decoded by initial level

    // Can accomodate 1 more stride in the level + concatenation of up to 2^4 tables
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    iasize        = AArch32.S2IASize(walkparams.t0sz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
```

## Library pseudocode for aarch32/translation/faults/AArch32.VAIsOutOfRange

```
// AArch32.VAIsOutOfRange()
// =====
// Check virtual address bits not resolved by translation are identical
// and of accepted value

boolean AArch32.VAIsOutOfRange(Regime regime, S1TTWParams walkparams, bits(32) va)
    if regime == Regime\_EL2 then
        // Input Address size
        iasize = AArch32.S1IASize(walkparams.t0sz);
        return walkparams.t0sz != '000' && !IsZero(va<31:iasize>);
    elseif walkparams.t1sz != '000' && walkparams.t0sz != '000' then
        // Lower range Input Address size
        lo_iasize = AArch32.S1IASize(walkparams.t0sz);
        // Upper range Input Address size
        up_iasize = AArch32.S1IASize(walkparams.t1sz);
        return !IsZero(va<31:lo_iasize>) && !IsOnes(va<31:up_iasize>);
    else
        return FALSE;
```

## Library pseudocode for aarch32/translation/tlbcontext/AArch32.GetS1TLBContext

```
// AArch32.GetS1TLBContext()
// =====
// Gather translation context for accesses with VA to match against TLB entries

TLBContext AArch32.GetS1TLBContext(Regime regime, SecurityState ss, bits(32) va)
    TLBContext tlbcontext;

    case regime of
        when Regime\_EL2 tlbcontext = AArch32.TLBContextEL2(va);
        when Regime\_EL10 tlbcontext = AArch32.TLBContextEL10(ss, va);
        when Regime\_EL30 tlbcontext = AArch32.TLBContextEL30(va);

    tlbcontext.includes_s1 = TRUE;
    // The following may be amended for EL1&0 Regime if caching of stage 2 is successful
    tlbcontext.includes_s2 = FALSE;
    return tlbcontext;
```

## Library pseudocode for aarch32/translation/tlbcontext/AArch32.GetS2TLBContext

```
// AArch32.GetS2TLBContext()
// =====
// Gather translation context for accesses with IPA to match against TLB entries

TLBContext AArch32.GetS2TLBContext(FullAddress ipa)
    assert ipa.paspace == PAS_NonSecure;

    TLBContext tlbcontext;

    tlbcontext.ss          = SS_NonSecure;
    tlbcontext.regime      = Regime_EL10;
    tlbcontext.ipaspace    = ipa.paspace;
    tlbcontext.vmid        = ZeroExtend(VTTBR.VMID, 16);
    tlbcontext.tg          = TGx_4KB;
    tlbcontext.includes_s1 = FALSE;
    tlbcontext.includes_s2 = TRUE;
    tlbcontext.ia          = ZeroExtend(ipa.address, 64);
    tlbcontext.cnp         = if HaveCommonNotPrivateTransExt() then VTTBR.CnP else '0';

    return tlbcontext;
```

## Library pseudocode for aarch32/translation/tlbcontext/AArch32.TLBContextEL10

```
// AArch32.TLBContextEL10()
// =====
// Gather translation context for accesses under EL10 regime
// (PL10 when EL3 is A64) to match against TLB entries

TLBContext AArch32.TLBContextEL10(SecurityState ss, bits(32) va)
    TLBContext tlbcontext;
    TTBCR_Type ttbcr;
    TTBR0_Type ttbr0;
    TTBR1_Type ttbr1;
    CONTEXTIDR_Type contextidr;

    if HaveAArch32EL(EL3) then
        ttbcr      = TTBCR_NS;
        ttbr0      = TTBR0_NS;
        ttbr1      = TTBR1_NS;
        contextidr = CONTEXTIDR_NS;
    else
        ttbcr      = TTBCR;
        ttbr0      = TTBR0;
        ttbr1      = TTBR1;
        contextidr = CONTEXTIDR;

    tlbcontext.ss      = ss;
    tlbcontext.regime = Regime\_EL10;

    if AArch32.EL2Enabled(ss) then
        tlbcontext.vmid = ZeroExtend(VTTBR.VMID, 16);

    if ttbcr.EAE == '1' then
        tlbcontext.asid = ZeroExtend(if ttbcr.A1 == '0' then ttbr0.ASID else ttbr1.ASID, 16);
    else
        tlbcontext.asid = ZeroExtend(contextidr.ASID, 16);

    tlbcontext.tg = TGx\_4KB;
    tlbcontext.ia = ZeroExtend(va, 64);

    if HaveCommonNotPrivateTransExt() && ttbcr.EAE == '1' then
        if AArch32.GetVARange(va, ttbcr.T0SZ, ttbcr.T1SZ) == VARange\_LOWER then
            tlbcontext.cnp = ttbr0.CnP;
        else
            tlbcontext.cnp = ttbr1.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

## Library pseudocode for aarch32/translation/tlbcontext/AArch32.TLBContextEL2

```
// AArch32.TLBContextEL2()
// =====
// Gather translation context for accesses under EL2 regime to match against TLB entries

TLBContext AArch32.TLBContextEL2(bits(32) va)
    TLBContext tlbcontext;

    tlbcontext.ss      = SS\_NonSecure;
    tlbcontext.regime = Regime\_EL2;
    tlbcontext.ia      = ZeroExtend(va, 64);
    tlbcontext.tg      = TGx\_4KB;
    tlbcontext.cnp      = if HaveCommonNotPrivateTransExt() then HTTBR.CnP else '0';

    return tlbcontext;
```

## Library pseudocode for aarch32/translation/tlbcontext/AArch32.TLBContextEL30

```
// AArch32.TLBContextEL30()
// =====
// Gather translation context for accesses under EL30 regime
// (PL10 in Secure state and EL3 is A32) to match against TLB entries

TLBContext AArch32.TLBContextEL30(bits(32) va)
    TLBContext tlbcontext;

    tlbcontext.ss      = SS\_Secure;
    tlbcontext.regime = Regime\_EL30;

    if TTBCR_S.EAE == '1' then
        tlbcontext.asid = ZeroExtend(if TTBCR_S.A1 == '0' then TTBR0_S.ASID else TTBR1_S.ASID, 16);
    else
        tlbcontext.asid = ZeroExtend(CONTEXTIDR_S.ASID, 16);

    tlbcontext.tg = TGx\_4KB;
    tlbcontext.ia = ZeroExtend(va, 64);

    if HaveCommonNotPrivateTransExt() && TTBCR_S.EAE == '1' then
        if AArch32.GetVARange(va, TTBCR_S.T0SZ, TTBCR_S.T1SZ) == VARange\_LOWER then
            tlbcontext.cnp = TTBR0_S.CnP;
        else
            tlbcontext.cnp = TTBR1_S.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

## Library pseudocode for aarch32/translation/translation/AArch32.EL2Enabled

```
// AArch32.EL2Enabled()
// =====
// Returns whether EL2 is enabled for the given Security State

boolean AArch32.EL2Enabled(SecurityState ss)
    if ss == SS\_Secure then
        if !(HaveEL(EL2) && HaveSecureEL2Ext()) then
            return FALSE;
        elsif HaveEL(EL3) then
            return SCR_EL3.EEL2 == '1';
        else
            return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
    else
        return HaveEL(EL2);
```

## Library pseudocode for aarch32/translation/translation/AArch32.FullTranslate

```
// AArch32.FullTranslate()
// =====
// Perform address translation as specified by VMSA-A32

AddressDescriptor AArch32.FullTranslate(bits(32) va, AccessDescriptor accdesc, boolean aligned)

    // Prepare fault fields in case a fault is detected
    FaultRecord fault = NoFault(accdesc);
    Regime regime = TranslationRegime(accdesc.el);

    // First Stage Translation
    AddressDescriptor ipa;
    if regime == Regime\_EL2 || TTBCR.EAE == '1' then
        (fault, ipa) = AArch32.S1TranslateLD(fault, regime, va, aligned, accdesc);
    else
        (fault, ipa, -) = AArch32.S1TranslateSD(fault, regime, va, aligned, accdesc);

    if fault.statuscode != Fault\_None then
        return CreateFaultyAddressDescriptor(ZeroExtend(va, 64), fault);

    if regime == Regime\_EL10 && EL2Enabled() then
        ipa.vaddress = ZeroExtend(va, 64);
        AddressDescriptor pa;
        (fault, pa) = AArch32.S2Translate(fault, ipa, aligned, accdesc);

        if fault.statuscode != Fault\_None then
            return CreateFaultyAddressDescriptor(ZeroExtend(va, 64), fault);
        else
            return pa;
    else
        return ipa;
```

## Library pseudocode for aarch32/translation/translation/AArch32.OutputDomain

```
// AArch32.OutputDomain()
// =====
// Determine the domain the translated output address

bits(2) AArch32.OutputDomain(Regime regime, bits(4) domain)
    bits(2) Dn;
    index = 2 * UInt(domain);
    if regime == Regime\_EL30 then
        Dn = DACR_S<index+1:index>;
    elsif HaveAArch32EL(EL3) then
        Dn = DACR_NS<index+1:index>;
    else
        Dn = DACR<index+1:index>;

    if Dn == '10' then
        // Reserved value maps to an allocated value
        (-, Dn) = ConstrainUnpredictableBits(Unpredictable\_RESDACR, 2);

    return Dn;
```





```

// AArch32.S1DisabledOutput()
// =====
// Flat map the VA to IPA/PA, depending on the regime, assigning default memory attributes

(FaultRecord, AddressDescriptor) AArch32.S1DisabledOutput(FaultRecord fault_in, Regime regime,
                                                         bits(32) va, boolean aligned,
                                                         AccessDescriptor accdesc)

FaultRecord fault = fault_in;
// No memory page is guarded when stage 1 address translation is disabled
SetInGuardedPage(FALSE);

MemoryAttributes memattrs;
bit default_cacheable;
if regime == Regime\_EL10 && AArch32.EL2Enabled(accdesc.ss) then
    if ELStateUsingAArch32(EL2, accdesc.ss == SS\_Secure) then
        default_cacheable = HCR.DC;
    else
        default_cacheable = HCR_EL2.DC;
else
    default_cacheable = '0';

if default_cacheable == '1' then
    // Use default cacheable settings
    memattrs.memtype = MemType\_Normal;
    memattrs.inner.attrs = MemAttr\_WB;
    memattrs.inner.hints = MemHint\_RWA;
    memattrs.outer.attrs = MemAttr\_WB;
    memattrs.outer.hints = MemHint\_RWA;
    memattrs.shareability = Shareability\_NSH;
    if (EL2Enabled() && !ELStateUsingAArch32(EL2, accdesc.ss == SS\_Secure) &&
        HaveMTE2Ext() && HCR_EL2.DCT == '1') then
        memattrs.tags = MemTag\_AllocationTagged;
    else
        memattrs.tags = MemTag\_Untagged;
    memattrs.xs = '0';
elseif accdesc.acctype == AccessType\_IFETCH then
    memattrs.memtype = MemType\_Normal;
    memattrs.shareability = Shareability\_OSH;
    memattrs.tags = MemTag\_Untagged;
    if AArch32.S1ICacheEnabled(regime) then
        memattrs.inner.attrs = MemAttr\_WT;
        memattrs.inner.hints = MemHint\_RA;
        memattrs.outer.attrs = MemAttr\_WT;
        memattrs.outer.hints = MemHint\_RA;
    else
        memattrs.inner.attrs = MemAttr\_NC;
        memattrs.outer.attrs = MemAttr\_NC;
    memattrs.xs = '1';
else
    // Treat memory region as Device
    memattrs.memtype = MemType\_Device;
    memattrs.device = DeviceType\_nGnRnE;
    memattrs.shareability = Shareability\_OSH;
    memattrs.tags = MemTag\_Untagged;
    memattrs.xs = '1';

bit ntlsmd;
if HaveTrapLoadStoreMultipleDeviceExt() then
    case regime of
        when Regime\_EL30 ntlsmd = SCTLR.S.nTLSMD;
        when Regime\_EL2 ntlsmd = HSCTLR.nTLSMD;
        when Regime\_EL10 ntlsmd = if HaveAArch32EL(EL3) then SCTLR_NS.nTLSMD else SCTLR.nTLSMD;
else
    ntlsmd = '1';

if AArch32.S1HasAlignmentFault(accdesc, aligned, ntlsmd, memattrs) then
    fault.statuscode = Fault\_Alignment;
    return (fault, AccessDescriptor UNKNOWN);

```

```

FullAddress oa;
oa.address = ZeroExtend(va, 56);
oa.paspace = if accdesc.ss == SS_Secure then PAS_Secure else PAS_NonSecure;
ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs);

return (fault, ipa);

```

## Library pseudocode for aarch32/translation/translation/AArch32.S1Enabled

```

// AArch32.S1Enabled()
// =====
// Returns whether stage 1 translation is enabled for the active translation regime

boolean AArch32.S1Enabled(Regime regime, SecurityState ss)
    if regime == Regime_EL2 then
        return HSCTLR.M == '1';
    elsif regime == Regime_EL30 then
        return SCTLR_S.M == '1';
    elsif !AArch32.EL2Enabled(ss) then
        return (if HaveAArch32EL(EL3) then SCTLR_NS.M else SCTLR.M) == '1';
    elsif ELStateUsingAArch32(EL2, ss == SS_Secure) then
        return HCR.<TGE,DC> == '00' && (if HaveAArch32EL(EL3) then SCTLR_NS.M else SCTLR.M) == '1';
    else
        return EL2Enabled() && HCR_EL2.<TGE,DC> == '00' && SCTLR.M == '1';

```

## Library pseudocode for aarch32/translation/translation/AArch32.S1TranslateLD

```
// AArch32.S1TranslateLD()
// =====
// Perform a stage 1 translation using long-descriptor format mapping VA to IPA/PA
// depending on the regime

(FaultRecord, AddressDescriptor) AArch32.S1TranslateLD(FaultRecord fault_in, Regime regime,
                                                    bits(32) va, boolean aligned,
                                                    AccessDescriptor accdesc)

FaultRecord fault = fault_in;

if !AArch32.S1Enabled(regime, accdesc.ss) then
    return AArch32.S1DisabledOutput(fault, regime, va, aligned, accdesc);

walkparams = AArch32.GetS1TTWParams(regime, va);

if AArch32.VAIsOutOfRange(regime, walkparams, va) then
    fault.level = 1;
    fault.statuscode = Fault\_Translation;
    return (fault, AddressDescriptor UNKNOWN);

TTWState walkstate;
(fault, walkstate) = AArch32.S1WalkLD(fault, regime, walkparams, accdesc, va);

if fault.statuscode != Fault\_None then
    return (fault, AddressDescriptor UNKNOWN);

SetInGuardedPage(FALSE); // AArch32-VMSEA does not guard any pages

if AArch32.S1HasAlignmentFault(accdesc, aligned, walkparams.ntlsmid, walkstate.memattrs) then
    fault.statuscode = Fault\_Alignment;
elseif AArch32.S1LDHasPermissionsFault(regime, walkparams,
                                         walkstate.permissions,
                                         walkstate.memattrs.memtype,
                                         walkstate.baseaddress.paspace,
                                         accdesc) then
    fault.statuscode = Fault\_Permission;

if fault.statuscode != Fault\_None then
    return (fault, AddressDescriptor UNKNOWN);

MemoryAttributes memattrs;
if ((accdesc.acctype == AccessType\_IFETCH &&
    (walkstate.memattrs.memtype == MemType\_Device || !AArch32.S1ICacheEnabled(regime))) ||
    (accdesc.acctype != AccessType\_IFETCH &&
    walkstate.memattrs.memtype == MemType\_Normal && !AArch32.S1DCacheEnabled(regime))) then
    // Treat memory attributes as Normal Non-Cacheable
    memattrs = NormalNCMemAttr();
    memattrs.xs = walkstate.memattrs.xs;
else
    memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime\_EL10 && AArch32.EL2Enabled(accdesc.ss) &&
    (if ELStateUsingAArch32(EL2, accdesc.ss==SS\_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    memattrs.shareability = walkstate.memattrs.shareability;
else
    memattrs.shareability = EffectiveShareability(memattrs);

// Output Address
oa = StageOA(ZeroExtend(va, 64), walkparams.d128, walkparams.tgx, walkstate);
ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs);

return (fault, ipa);
```



```

// AArch32.S1TranslateSD()
// =====
// Perform a stage 1 translation using short-descriptor format mapping VA to IPA/PA
// depending on the regime

(FaultRecord, AddressDescriptor, SDType) AArch32.S1TranslateSD(FaultRecord fault_in, Regime regime,
                                                                bits(32) va, boolean aligned,
                                                                AccessDescriptor accdesc)

FaultRecord fault = fault_in;

if !AArch32.S1Enabled(regime, accdesc.ss) then
    AddressDescriptor ipa;
    (fault, ipa) = AArch32.S1DisabledOutput(fault, regime, va, aligned, accdesc);
    return (fault, ipa, SDType UNKNOWN);

TTWState walkstate;
(fault, walkstate) = AArch32.S1WalkSD(fault, regime, accdesc, va);

if fault.statuscode != Fault\_None then
    return (fault, AddressDescriptor UNKNOWN, SDType UNKNOWN);

domain = AArch32.OutputDomain(regime, walkstate.domain);
SetInGuardedPage(FALSE); // AArch32-VMSA does not guard any pages

bit ntlsmd;
if HaveTrapLoadStoreMultipleDeviceExt() then
    case regime of
        when Regime\_EL30 ntlsmd = SCTLR_S.nTLSMD;
        when Regime\_EL10 ntlsmd = if HaveAArch32EL(EL3) then SCTLR_NS.nTLSMD else SCTLR.nTLSMD;
    else
        ntlsmd = '1';

if AArch32.S1HasAlignmentFault(accdesc, aligned, ntlsmd, walkstate.memattrs) then
    fault.statuscode = Fault\_Alignment;
elsif (!(accdesc.acctype IN {AccessType\_IC, AccessType\_DC}) &&
        domain == Domain\_NoAccess) then
    fault.statuscode = Fault\_Domain;
elsif domain == Domain\_Client then
    if AArch32.S1SDHasPermissionsFault(regime, walkstate.permissions,
                                         walkstate.memattrs.memtype,
                                         walkstate.baseaddress.paspace,
                                         accdesc) then
        fault.statuscode = Fault\_Permission;

if fault.statuscode != Fault\_None then
    fault.domain = walkstate.domain;
    return (fault, AddressDescriptor UNKNOWN, walkstate.sdftype);

MemoryAttributes memattrs;
if ((accdesc.acctype == AccessType\_IFETCH &&
    (walkstate.memattrs.memtype == MemType\_Device || !AArch32.S1ICacheEnabled(regime))) ||
    (accdesc.acctype != AccessType\_IFETCH &&
    walkstate.memattrs.memtype == MemType\_Normal && !AArch32.S1DCacheEnabled(regime))) then
    // Treat memory attributes as Normal Non-Cacheable
    memattrs = NormalNCMemAttr();
    memattrs.xs = walkstate.memattrs.xs;
else
    memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime\_EL10 && AArch32.EL2Enabled(accdesc.ss) &&
    (if ELStateUsingAArch32(EL2, accdesc.ss==SS\_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    memattrs.shareability = walkstate.memattrs.shareability;
else
    memattrs.shareability = EffectiveShareability(memattrs);

// Output Address

```

```
oa = AArch32.SDStageOA(walkstate.baseaddress, va, walkstate.sdftype);  
ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs);  
  
return (fault, ipa, walkstate.sdftype);
```

## Library pseudocode for aarch32/translation/translation/AArch32.S2Translate

```
// AArch32.S2Translate()
// =====
// Perform a stage 2 translation mapping an IPA to a PA

(FaultRecord, AddressDescriptor) AArch32.S2Translate(FaultRecord fault_in, AddressDescriptor ipa,
                                                    boolean aligned, AccessDescriptor accdesc)

    FaultRecord fault = fault_in;
    assert IsZero(ipa.paddress.address<55:40>);

    if !ELStateUsingAArch32(EL2, accdesc.ss == SS_Secure) then
        slaarch64 = FALSE;
        return AArch64.S2Translate(fault, ipa, slaarch64, aligned, accdesc);

    // Prepare fault fields in case a fault is detected
    fault.statuscode = Fault_None;
    fault.secondstage = TRUE;
    fault.s2fslwalk = accdesc.acctype == AccessType_TTW;
    fault.ipaddress = ipa.paddress;

    walkparams = AArch32.GetS2TTWParams();

    if walkparams.vm == '0' then
        // Stage 2 is disabled
        return (fault, ipa);

    if AArch32.IPAIsOutOfRange(walkparams, ipa.paddress.address<39:0>) then
        fault.statuscode = Fault_Translation;
        fault.level = 1;
        return (fault, AddressDescriptor UNKNOWN);

    TTWState walkstate;
    (fault, walkstate) = AArch32.S2Walk(fault, walkparams, accdesc, ipa);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    if AArch32.S2HasAlignmentFault(accdesc, aligned, walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elseif AArch32.S2HasPermissionsFault(walkparams,
                                         walkstate.permissions,
                                         walkstate.memattrs.memtype,
                                         accdesc) then
        fault.statuscode = Fault_Permission;
    MemoryAttributes s2_memattrs;
    if ((accdesc.acctype == AccessType_TTW &&
        walkstate.memattrs.memtype == MemType_Device) ||
        (accdesc.acctype == AccessType_IFETCH &&
        (walkstate.memattrs.memtype == MemType_Device || HCR2.ID == '1')) ||
        (accdesc.acctype != AccessType_IFETCH &&
        walkstate.memattrs.memtype == MemType_Normal && HCR2.CD == '1')) then
        // Treat memory attributes as Normal Non-Cacheable
        s2_memattrs = NormalNCMemAttr();
        s2_memattrs.xs = walkstate.memattrs.xs;
    else
        s2_memattrs = walkstate.memattrs;

    s2aarch64 = FALSE;
    memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs, s2aarch64);
    ipa_64 = ZeroExtend(ipa.paddress.address<39:0>, 64);
    // Output Address
    oa = StageOA(ipa_64, walkparams.d128, walkparams.tgx, walkstate);
    pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs);

    return (fault, pa);
```

## Library pseudocode for aarch32/translation/translation/AArch32.SDStageOA

```
// AArch32.SDStageOA()
// =====
// Given the final walk state of a short-descriptor translation walk,
// map the untranslated input address bits to the base output address

FullAddress AArch32.SDStageOA(FullAddress baseaddress, bits(32) va, SDFType sdftype)
integer tsize;
case sdftype of
    when SDFType_SmallPage      tsize = 12;
    when SDFType_LargePage     tsize = 16;
    when SDFType_Section       tsize = 20;
    when SDFType_Supersection  tsize = 24;

// Output Address
FullAddress oa;
oa.address = baseaddress.address<55:tsize>:va<tsize-1:0>;
oa.paspace = baseaddress.paspace;
return oa;
```

## Library pseudocode for aarch32/translation/translation/AArch32.TranslateAddress

```
// AArch32.TranslateAddress()
// =====
// Main entry point for translating an address

AddressDescriptor AArch32.TranslateAddress(bits(32) va, AccessDescriptor accdesc,
                                           boolean aligned, integer size)

Regime regime = TranslationRegime(PSTATE.EL);
if !RegimeUsingAArch32(regime) then
    return AArch64.TranslateAddress(ZeroExtend(va, 64), accdesc, aligned, size);

AddressDescriptor result = AArch32.FullTranslate(va, accdesc, aligned);

if !IsFault(result) then
    result.fault = AArch32.CheckDebug(va, accdesc, size);

// Update virtual address for abort functions
result.vaddress = ZeroExtend(va, 64);

return result;
```

## Library pseudocode for aarch32/translation/walk/AArch32.DecodeDescriptorTypeLD

```
// AArch32.DecodeDescriptorTypeLD()
// =====
// Determine whether the long-descriptor is a page, block or table

DescriptorType AArch32.DecodeDescriptorTypeLD(bits(64) descriptor, integer level)
if descriptor<1:0> == '11' && level == FINAL_LEVEL then
    return DescriptorType_Leaf;
elseif descriptor<1:0> == '11' then
    return DescriptorType_Table;
elseif descriptor<1:0> == '01' && level != FINAL_LEVEL then
    return DescriptorType_Leaf;
else
    return DescriptorType_Invalid;
```



## Library pseudocode for aarch32/translation/walk/AArch32.DecodeDescriptorTypeSD

```
// AArch32.DecodeDescriptorTypeSD()
// =====
// Determine the type of the short-descriptor

SDFType AArch32.DecodeDescriptorTypeSD(bits(32) descriptor, integer level)
    if level == 1 && descriptor<1:0> == '01' then
        return SDFType\_Table;
    elsif level == 1 && descriptor<18,1> == '01' then
        return SDFType\_Section;
    elsif level == 1 && descriptor<18,1> == '11' then
        return SDFType\_Supersection;
    elsif level == 2 && descriptor<1:0> == '01' then
        return SDFType\_LargePage;
    elsif level == 2 && descriptor<1:0> IN {'1x'} then
        return SDFType\_SmallPage;
    else
        return SDFType\_Invalid;
```

## Library pseudocode for aarch32/translation/walk/AArch32.S1IASize

```
// AArch32.S1IASize()
// =====
// Retrieve the number of bits containing the input address for stage 1 translation

integer AArch32.S1IASize(bits(3) txsz)
    return 32 - UInt(txsz);
```



```

// AArch32.S1WalkLD()
// =====
// Traverse stage 1 translation tables in long format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S1WalkLD(FaultRecord fault_in, Regime regime,
                                         S1TTWParams walkparams, AccessDescriptor accdesc,
                                         bits(32) va)

FaultRecord fault = fault_in;
bits(3) txsz;
bits(64) ttbr;
bit epd;
VARange varange;
if regime == Regime\_EL2 then
    ttbr = HTTBR;
    txsz = walkparams.t0sz;
    varange = VARange\_LOWER;
else
    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    bits(64) ttbr0;
    bits(64) ttbr1;
    TTBCR_Type ttbcr;
    if regime == Regime\_EL30 then
        ttbcr = TTBCR_S;
        ttbr0 = TTBR0_S;
        ttbr1 = TTBR1_S;
    elseif HaveAArch32EL(EL3) then
        ttbcr = TTBCR_NS;
        ttbr0 = TTBR0_NS;
        ttbr1 = TTBR1_NS;
    else
        ttbcr = TTBCR;
        ttbr0 = TTBR0;
        ttbr1 = TTBR1;

    assert ttbcr.EAE == '1';
    if varange == VARange\_LOWER then
        txsz = walkparams.t0sz;
        ttbr = ttbr0;
        epd = ttbcr.EPD0;
    else
        txsz = walkparams.t1sz;
        ttbr = ttbr1;
        epd = ttbcr.EPD1;

if regime != Regime\_EL2 && epd == '1' then
    fault.level = 1;
    fault.statuscode = Fault\_Translation;
    return (fault, TTWState UNKNOWN);

// Input Address size
iasize = AArch32.S1IASize(txsz);
granulebits = TGxGranuleBits(walkparams.tgx);
stride = granulebits - 3;
startlevel = FINAL\_LEVEL - (((iasize-1) - granulebits) DIV stride);
levels = FINAL\_LEVEL - startlevel;

if !IsZero(ttbr<47:40>) then
    fault.statuscode = Fault\_AddressSize;
    fault.level = 0;
    return (fault, TTWState UNKNOWN);

FullAddress baseaddress;
baselsb = (iasize - (levels*stride + granulebits)) + 3;
baseaddress.paspace = if accdesc.ss == SS\_Secure then PAS\_Secure else PAS\_NonSecure;
baseaddress.address = ZeroExtend(ttbr<39:baselsb>:Zeros(baselsb), 56);

TTWState walkstate;
walkstate.baseaddress = baseaddress;
walkstate.level = startlevel;
walkstate.istable = TRUE;

```

```

// In regimes that support global and non-global translations, translation
// table entries from lookup levels other than the final level of lookup
// are treated as being non-global
walkstate.nG          = if HasUnprivileged(regime) then '1' else '0';
walkstate.memattrs     = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
walkstate.permissions.ap_table = '00';
walkstate.permissions.xn_table = '0';
walkstate.permissions.pxn_table = '0';

indexmsb = iasize - 1;
bits(64) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = ZeroExtend(va, 64);

if !AArch32.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime\_EL10 && AArch32.EL2Enabled(accdesc.ss) &&
    (if ELStateUsingAArch32(EL2, accdesc.ss==SS\_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

integer indexlsb;
DescriptorType desctype;
repeat
    fault.level = walkstate.level;
    indexlsb = (FINAL\_LEVEL - walkstate.level)*stride + granulebits;
    bits(40) index = ZeroExtend(va<indexmsb:indexlsb>:'000', 40);

    walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index, 56);
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    boolean toplevel = walkstate.level == startlevel;
    AccessDescriptor walkaccess = CreateAccDescS1TTW(toplevel, varange, accdesc);
    // If there are two stages of translation, then the first stage table walk addresses
    // are themselves subject to translation
    if regime == Regime\_EL10 && AArch32.EL2Enabled(accdesc.ss) then
        s2aligned = TRUE;
        (s2fault, s2walkaddress) = AArch32.S2Translate(fault, walkaddress, s2aligned,
                                                    walkaccess);

        // Check for a fault on the stage 2 walk
        if s2fault.statuscode != Fault\_None then
            return (s2fault, TTWState UNKNOWN);

        (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, walkaccess,
                                                    fault, 64);
    else
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, walkaccess,
                                                    fault, 64);

    if fault.statuscode != Fault\_None then
        return (fault, TTWState UNKNOWN);

    desctype = AArch32.DecodeDescriptorTypeLD(descriptor, walkstate.level);

    case desctype of
        when DescriptorType\_Table
            if !IsZero(descriptor<47:40>) then
                fault.statuscode = Fault\_AddressSize;
                return (fault, TTWState UNKNOWN);

    walkstate.baseaddress.address = ZeroExtend(descriptor<39:12>:Zeros(12), 56);

```

```

        if walkstate.baseaddress.paspace == PAS\_Secure && descriptor<63> == '1' then
            walkstate.baseaddress.paspace = PAS\_NonSecure;

        if walkparams.hpd == '0' then
            walkstate.permissions.xn_table = (walkstate.permissions.xn_table OR
                                              descriptor<60>);
            walkstate.permissions.ap_table = (walkstate.permissions.ap_table OR
                                              descriptor<62:61>);
            walkstate.permissions.pxn_table = (walkstate.permissions.pxn_table OR
                                              descriptor<59>);

        walkstate.level = walkstate.level + 1;
        indexmsb = indexlsb - 1;

        when DescriptorType Invalid
            fault.statuscode = Fault\_Translation;
            return (fault, TTWState UNKNOWN);

        when DescriptorType Leaf
            walkstate.istable = FALSE;

until desctype == DescriptorType\_Leaf;

// Check the output address is inside the supported range
if !IsZero(descriptor<47:40>) then
    fault.statuscode = Fault\_AddressSize;
    return (fault, TTWState UNKNOWN);

// Check the access flag
if descriptor<10> == '0' then
    fault.statuscode = Fault\_AccessFlag;
    return (fault, TTWState UNKNOWN);

walkstate.permissions.xn = descriptor<54>;
walkstate.permissions.pxn = descriptor<53>;
walkstate.permissions.ap = descriptor<7:6>:'1';
walkstate.contiguous = descriptor<52>;
if regime == Regime\_EL2 then
    // All EL2 regime accesses are treated as Global
    walkstate.nG = '0';
elseif accdesc.ss == SS\_Secure && walkstate.baseaddress.paspace == PAS\_NonSecure then
    // When a PE is using the Long-descriptor translation table format,
    // and is in Secure state, a translation must be treated as non-global,
    // regardless of the value of the nG bit,
    // if NSTable is set to 1 at any level of the translation table walk.
    walkstate.nG = '1';
else
    walkstate.nG = descriptor<11>;

walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb), 56);
if walkstate.baseaddress.paspace == PAS\_Secure && descriptor<5> == '1' then
    walkstate.baseaddress.paspace = PAS\_NonSecure;

memattr = descriptor<4:2>;
sh = descriptor<9:8>;
attr = AArch32.MAIRAttr(UInt(memattr), walkparams.mair);
slaarch64 = FALSE;
walkstate.memattrs = S1DecodeMemAttrs(attr, sh, slaarch64, walkparams);

return (fault, walkstate);

```



```

// AArch32.S1WalkSD()
// =====
// Traverse stage 1 translation tables in short format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S1WalkSD(FaultRecord fault_in, Regime regime,
                                           AccessDescriptor accdesc, bits(32) va)

    FaultRecord fault = fault_in;
    SCTLr_Type sctlr;
    TTBCr_Type ttbcr;
    TTBR0_Type ttbr0;
    TTBR1_Type ttbr1;
    // Determine correct translation control registers to use.
    if regime == Regime EL30 then
        sctlr = SCTLr_S;
        ttbcr = TTBCr_S;
        ttbr0 = TTBR0_S;
        ttbr1 = TTBR1_S;
    elseif HaveAArch32EL\(EL3\) then
        sctlr = SCTLr_NS;
        ttbcr = TTBCr_NS;
        ttbr0 = TTBR0_NS;
        ttbr1 = TTBR1_NS;
    else
        sctlr = SCTLr;
        ttbcr = TTBCr;
        ttbr0 = TTBR0;
        ttbr1 = TTBR1;

    assert ttbcr.EAE == '0';
    ee = sctlr.EE;
    afe = sctlr.AFE;
    tre = sctlr.TRE;
    n = UInt(ttbcr.N);
    bits(32) ttb;
    bits(1) pd;
    bits(2) irgn;
    bits(2) rgn;
    bits(1) s;
    bits(1) nos;
    VArange varange;
    if n == 0 || IsZero(va<31:(32-n)>) then
        ttb = ttbr0.TTB0:Zeros(7);
        pd = ttbcr.PD0;
        irgn = ttbr0.IRGN;
        rgn = ttbr0.RGN;
        s = ttbr0.S;
        nos = ttbr0.NOS;
        varange = VArange LOWER;
    else
        n = 0; // TTBR1 translation always treats N as 0
        ttb = ttbr1.TTB1:Zeros(7);
        pd = ttbcr.PD1;
        irgn = ttbr1.IRGN;
        rgn = ttbr1.RGN;
        s = ttbr1.S;
        nos = ttbr1.NOS;
        varange = VArange UPPER;

    // Check if Translation table walk disabled for translations with this Base register.
    if pd == '1' then
        fault.level = 1;
        fault.statuscode = Fault Translation;
        return (fault, TTWState UNKNOWN);

    FullAddress baseaddress;
    baseaddress.paspace = if accdesc.ss == SS Secure then PAS Secure else PAS NonSecure;
    baseaddress.address = ZeroExtend(ttb<31:14-n>:Zeros(14-n), 56);

    constant integer startlevel = 1;
    TTWState walkstate;

```

```

walkstate.baseaddress = baseaddress;
// In regimes that support global and non-global translations, translation
// table entries from lookup levels other than the final level of lookup
// are treated as being non-global. Translations in Short-Descriptor Format
// always support global & non-global translations.
walkstate.nG          = '1';
walkstate.memattrs    = WalkMemAttrs(s:nos, irgn, rgn);
walkstate.level       = startlevel;
walkstate.istable     = TRUE;

bits(4) domain;
bits(32) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = ZeroExtend(va, 64);

if !AArch32.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime\_EL10 && AArch32.EL2Enabled(accdesc.ss) &&
    (if ELStateUsingAArch32(EL2, accdesc.ss==SS\_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

bit nG;
bit ns;
bit pxn;
bits(3) ap;
bits(3) tex;
bit c;
bit b;
bit xn;
repeat
    fault.level = walkstate.level;

    bits(32) index;
    if walkstate.level == 1 then
        index = ZeroExtend(va<31-n:20>:'00', 32);
    else
        index = ZeroExtend(va<19:12>:'00', 32);

    walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index,
                                                                    56);
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    boolean toplevel = walkstate.level == startlevel;
    AccessDescriptor walkaccess = CreateAccDescS1TTW(toplevel, varange, accdesc);
    if regime == Regime\_EL10 && AArch32.EL2Enabled(accdesc.ss) then
        s2aligned = TRUE;
        (s2fault, s2walkaddress) = AArch32.S2Translate(fault, walkaddress, s2aligned,
                                                    walkaccess);

        if s2fault.statuscode != Fault\_None then
            return (s2fault, TTWState UNKNOWN);

        (fault, descriptor) = FetchDescriptor(ee, s2walkaddress, walkaccess, fault, 32);
    else
        (fault, descriptor) = FetchDescriptor(ee, walkaddress, walkaccess, fault, 32);

    if fault.statuscode != Fault\_None then
        return (fault, TTWState UNKNOWN);

    walkstate.sdftype = AArch32.DecodeDescriptorTypeSD(descriptor, walkstate.level);

```



```

case walkstate.sdftype of
  when SDFTYPE Invalid
    fault.domain = domain;
    fault.statuscode = Fault Translation;
    return (fault, TTWState UNKNOWN);

  when SDFTYPE Table
    domain = descriptor<8:5>;
    ns = descriptor<3>;
    pxn = descriptor<2>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<31:10>:Zeros (10),
                                                56);

    walkstate.level = 2;

  when SDFTYPE SmallPage
    nG = descriptor<11>;
    s = descriptor<10>;
    ap = descriptor<9,5:4>;
    tex = descriptor<8:6>;
    c = descriptor<3>;
    b = descriptor<2>;
    xn = descriptor<0>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<31:12>:Zeros (12),
                                                56);

    walkstate.istable = FALSE;

  when SDFTYPE LargePage
    xn = descriptor<15>;
    tex = descriptor<14:12>;
    nG = descriptor<11>;
    s = descriptor<10>;
    ap = descriptor<9,5:4>;
    c = descriptor<3>;
    b = descriptor<2>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<31:16>:Zeros (16),
                                                56);

    walkstate.istable = FALSE;

  when SDFTYPE Section
    ns = descriptor<19>;
    nG = descriptor<17>;
    s = descriptor<16>;
    ap = descriptor<15,11:10>;
    tex = descriptor<14:12>;
    domain = descriptor<8:5>;
    xn = descriptor<4>;
    c = descriptor<3>;
    b = descriptor<2>;
    pxn = descriptor<0>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<31:20>:Zeros (20),
                                                56);

    walkstate.istable = FALSE;

  when SDFTYPE Supersection
    ns = descriptor<19>;
    nG = descriptor<17>;
    s = descriptor<16>;
    ap = descriptor<15,11:10>;
    tex = descriptor<14:12>;
    xn = descriptor<4>;
    c = descriptor<3>;
    b = descriptor<2>;
    pxn = descriptor<0>;
    domain = '0000';

```

```

        walkstate.baseaddress.address = ZeroExtend(descriptor<8:5,23:20,31:24>:Zeros(24),
                                                    56);

        walkstate.istable = FALSE;

until walkstate.sdftype != SDFType\_Table;

if afe == '1' && ap<0> == '0' then
    fault.domain      = domain;
    fault.statuscode = Fault\_AccessFlag;
    return (fault, TTWState UNKNOWN);

// Decode the TEX, C, B and S bits to produce target memory attributes
if tre == '1' then
    walkstate.memattrs = AArch32.RemappedTEXDecode(regime, tex, c, b, s);
elsif RemapRegsHaveResetValues() then
    walkstate.memattrs = AArch32.DefaultTEXDecode(tex, c, b, s);
else
    walkstate.memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;

walkstate.permissions.ap = ap;
walkstate.permissions.xn = xn;
walkstate.permissions.pxn = pxn;
walkstate.domain = domain;
walkstate.nG      = nG;

if accdesc.ss == SS\_Secure && ns == '0' then
    walkstate.baseaddress.paspace = PAS\_Secure;
else
    walkstate.baseaddress.paspace = PAS\_NonSecure;

return (fault, walkstate);

```

#### Library pseudocode for aarch32/translation/walk/AArch32.S2IASize

```

// AArch32.S2IASize()
// =====
// Retrieve the number of bits containing the input address for stage 2 translation

integer AArch32.S2IASize(bits(4) t0sz)
    return 32 - SInt(t0sz);

```

#### Library pseudocode for aarch32/translation/walk/AArch32.S2StartLevel

```

// AArch32.S2StartLevel()
// =====
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch32.S2StartLevel(bits(2) sl0)
    return 2 - UInt(sl0);

```



```

// AArch32.S2Walk()
// =====
// Traverse stage 2 translation tables in long format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S2Walk(FaultRecord fault_in, S2TTWParams walkparams,
                                         AccessDescriptor accdesc, AddressDescriptor ipa)

    FaultRecord fault = fault_in;

    if walkparams.sl0 IN {'1x'} || AArch32.S2InconsistentSL(walkparams) then
        fault.statuscode = Fault\_Translation;
        fault.level      = 1;
        return (fault, TTWState UNKNOWN);

    // Input Address size
    iasize      = AArch32.S2IASize(walkparams.t0sz);
    startlevel  = AArch32.S2StartLevel(walkparams.sl0);
    levels      = FINAL\_LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;

    if !IsZero(VTBR<47:40>) then
        fault.statuscode = Fault\_AddressSize;
        fault.level      = 0;
        return (fault, TTWState UNKNOWN);

    FullAddress baseaddress;
    baselsb = (iasize - (levels*stride + granulebits)) + 3;
    baseaddress.paspace = PAS\_NonSecure;
    baseaddress.address = ZeroExtend(VTBR<39:baselsb>:Zeros(baselsb), 56);

    TTWState walkstate;
    walkstate.baseaddress = baseaddress;
    walkstate.level       = startlevel;
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn,
                                         walkparams.orgn);

    indexmsb = iasize - 1;
    bits(64) descriptor;
    AccessDescriptor walkaccess = CreateAccDescS2TTW(accdesc);
    AddressDescriptor walkaddress;

    walkaddress.vaddress = ipa.vaddress;

    if HCR2.CD == '1' then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;

    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

    integer indexlsb;
    DescriptorType desctype;
    repeat
        fault.level = walkstate.level;

        indexlsb = (FINAL\_LEVEL - walkstate.level)*stride + granulebits;
        bits(40) index = ZeroExtend(ipa.paddress.address<indexmsb:indexlsb>:'000', 40);

        walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index, 56);
        walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, walkaccess, fault, 64);

        if fault.statuscode != Fault\_None then
            return (fault, TTWState UNKNOWN);

        desctype = AArch32.DecodeDescriptorTypeLD(descriptor, walkstate.level);

```

```

case desctype of
  when DescriptorType Table
    if !IsZero(descriptor<47:40>) then
      fault.statuscode = Fault AddressSize;
      return (fault, TTWState UNKNOWN);

      walkstate.baseaddress.address = ZeroExtend(descriptor<39:12>:Zeros(12), 56);
      walkstate.level = walkstate.level + 1;
      indexmsb = indexlsb - 1;

  when DescriptorType Invalid
    fault.statuscode = Fault Translation;
    return (fault, TTWState UNKNOWN);

  when DescriptorType Leaf
    walkstate.istable = FALSE;

until desctype IN {DescriptorType Leaf};

// Check the output address is inside the supported range
if !IsZero(descriptor<47:40>) then
  fault.statuscode = Fault AddressSize;
  return (fault, TTWState UNKNOWN);

// Check the access flag
if descriptor<10> == '0' then
  fault.statuscode = Fault AccessFlag;
  return (fault, TTWState UNKNOWN);

// Unpack the descriptor into address and upper and lower block attributes
walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb), 56);

walkstate.permissions.s2ap = descriptor<7:6>;
walkstate.permissions.s2xn = descriptor<54>;
if HaveExtendedExecuteNeverExt() then
  walkstate.permissions.s2xnx = descriptor<53>;
else
  walkstate.permissions.s2xnx = '0';

memattr = descriptor<5:2>;
sh      = descriptor<9:8>;
s2aarch64 = FALSE;
walkstate.memattrs = S2DecodeMemAttrs(memattr, sh, s2aarch64);
walkstate.contiguous = descriptor<52>;

return (fault, walkstate);

```

### Library pseudocode for aarch32/translation/walk/AArch32.TranslationSizeSD

```

// AArch32.TranslationSizeSD()
// =====
// Determine the size of the translation

integer AArch32.TranslationSizeSD(SDFType sdftype)
  integer tsize;
  case sdftype of
    when SDFType SmallPage      tsize = 12;
    when SDFType LargePage     tsize = 16;
    when SDFType Section       tsize = 20;
    when SDFType Supersection  tsize = 24;

  return tsize;

```

### Library pseudocode for aarch32/translation/walk/RemapRegsHaveResetValues

```

// RemapRegsHaveResetValues()
// =====

boolean RemapRegsHaveResetValues();

```

## Library pseudocode for aarch32/translation/walkparams/AArch32.GetS1TTWParams

```
// AArch32.GetS1TTWParams()
// =====
// Returns stage 1 translation table walk parameters from respective controlling
// System registers.

S1TTWParams AArch32.GetS1TTWParams(Regime regime, bits(32) va)
    S1TTWParams walkparams;

    case regime of
        when Regime\_EL2 walkparams = AArch32.S1TTWParamsEL2();
        when Regime\_EL10 walkparams = AArch32.S1TTWParamsEL10(va);
        when Regime\_EL30 walkparams = AArch32.S1TTWParamsEL30(va);

    return walkparams;
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.GetS2TTWParams

```
// AArch32.GetS2TTWParams()
// =====
// Gather walk parameters for stage 2 translation

S2TTWParams AArch32.GetS2TTWParams()
    S2TTWParams walkparams;

    walkparams.tgx = TGx\_4KB;
    walkparams.s = VTCR.S;
    walkparams.t0sz = VTCR.T0SZ;
    walkparams.sl0 = VTCR.SL0;
    walkparams.irgn = VTCR.IRGN0;
    walkparams.orgn = VTCR.ORGNO;
    walkparams.sh = VTCR.SH0;
    walkparams.ee = HSCTLR.EE;
    walkparams.ptw = HCR.PTW;
    walkparams.vm = HCR.VM OR HCR.DC;

    // VTCR.S must match VTCR.T0SZ[3]
    if walkparams.s != walkparams.t0sz<3> then
        (-, walkparams.t0sz) = ConstrainUnpredictableBits(Unpredictable\_RESVTCRS, 4);

    return walkparams;
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.GetVARange

```
// AArch32.GetVARange()
// =====
// Select the translation base address for stage 1 long-descriptor walks

VARange AArch32.GetVARange(bits(32) va, bits(3) t0sz, bits(3) t1sz)
    // Lower range Input Address size
    lo_iasize = AArch32.S1IASize(t0sz);
    // Upper range Input Address size
    up_iasize = AArch32.S1IASize(t1sz);

    if t1sz == '000' && t0sz == '000' then
        return VARange\_LOWER;
    elsif t1sz == '000' then
        return if IsZero(va<31:lo_iasize>) then VARange\_LOWER else VARange\_UPPER;
    elsif t0sz == '000' then
        return if IsOnes(va<31:up_iasize>) then VARange\_UPPER else VARange\_LOWER;
    elsif IsZero(va<31:lo_iasize>) then
        return VARange\_LOWER;
    elsif IsOnes(va<31:up_iasize>) then
        return VARange\_UPPER;
    else
        // Will be reported as a Translation Fault
        return VARange UNKNOWN;
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.S1DCacheEnabled

```
// AArch32.S1DCacheEnabled()
// =====
// Determine cacheability of stage 1 data accesses

boolean AArch32.S1DCacheEnabled(Regime regime)
    case regime of
        when Regime\_EL30 return SCTLR_S.C == '1';
        when Regime\_EL2  return HSCTLR.C == '1';
        when Regime\_EL10 return (if HaveAArch32EL(EL3) then SCTLR_NS.C else SCTLR.C) == '1';
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.S1ICacheEnabled

```
// AArch32.S1ICacheEnabled()
// =====
// Determine cacheability of stage 1 instruction fetches

boolean AArch32.S1ICacheEnabled(Regime regime)
    case regime of
        when Regime\_EL30 return SCTLR_S.I == '1';
        when Regime\_EL2  return HSCTLR.I == '1';
        when Regime\_EL10 return (if HaveAArch32EL(EL3) then SCTLR_NS.I else SCTLR.I) == '1';
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.S1TTWParamsEL10

```
// AArch32.S1TTWParamsEL10()
// =====
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled).

S1TTWParams AArch32.S1TTWParamsEL10(bits(32) va)
    bits(64) mair;
    bit sif;
    TTBCR_Type ttbcr;
    TTBCR2_Type ttbcr2;
    SCTLR_Type sctlr;

    if ELUsingAArch32\(EL3\) then
        ttbcr = TTBCR_NS;
        ttbcr2 = TTBCR2_NS;
        sctlr = SCTLR_NS;
        mair = MAIR1_NS:MAIR0_NS;
        sif = SCR.SIF;
    else
        ttbcr = TTBCR;
        ttbcr2 = TTBCR2;
        sctlr = SCTLR;
        mair = MAIR1:MAIR0;
        sif = if HaveEL\(EL3\) then SCR_EL3.SIF else '0';

    assert ttbcr.EAE == '1';
    S1TTWParams walkparams;

    walkparams.t0sz = ttbcr.T0SZ;
    walkparams.t1sz = ttbcr.T1SZ;
    walkparams.ee = sctlr.EE;
    walkparams.wxn = sctlr.WXN;
    walkparams.uwxn = sctlr.UWXN;
    walkparams.ntlsmid = if HaveTrapLoadStoreMultipleDeviceExt\(\) then sctlr.nTLSMD else '1';
    walkparams.mair = mair;
    walkparams.sif = sif;

    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    if varange == VARange\_LOWER then
        walkparams.sh = ttbcr.SH0;
        walkparams.irgn = ttbcr.IRGN0;
        walkparams.orgn = ttbcr.ORGNO;
        walkparams.hpd = if AArch32.HaveHPDExt\(\) then ttbcr.T2E AND ttbcr2.HPD0 else '0';
    else
        walkparams.sh = ttbcr.SH1;
        walkparams.irgn = ttbcr.IRGN1;
        walkparams.orgn = ttbcr.ORGNI;
        walkparams.hpd = if AArch32.HaveHPDExt\(\) then ttbcr.T2E AND ttbcr2.HPD1 else '0';

    return walkparams;
```



## Library pseudocode for aarch32/translation/walkparams/AArch32.S1TTWParamsEL2

```
// AArch32.S1TTWParamsEL2()
// =====
// Gather stage 1 translation table walk parameters for EL2 regime

S1TTWParams AArch32.S1TTWParamsEL2()
    S1TTWParams walkparams;

    walkparams.tgx = TGx\_4KB;
    walkparams.t0sz = HTCR.T0SZ;
    walkparams.irgn = HTCR.SH0;
    walkparams.orgn = HTCR.IRGN0;
    walkparams.sh = HTCR.ORGNO;
    walkparams.hpd = if AArch32.HaveHPDExt() then HTCR.HPD else '0';
    walkparams.ee = HSCTLR.EE;
    walkparams.wxn = HSCTLR.WXN;
    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmid = HSCTLR.nTlSMID;
    else
        walkparams.ntlsmid = '1';

    walkparams.mair = HMAIR1:HMAIR0;

    return walkparams;
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.S1TTWParamsEL30

```
// AArch32.S1TTWParamsEL30()
// =====
// Gather stage 1 translation table walk parameters for EL3&0 regime

S1TTWParams AArch32.S1TTWParamsEL30(bits(32) va)
    assert TTBCR_S.EAE == '1';
    S1TTWParams walkparams;

    walkparams.t0sz = TTBCR_S.T0SZ;
    walkparams.t1sz = TTBCR_S.T1SZ;
    walkparams.ee = SCTLR_S.EE;
    walkparams.wxn = SCTLR_S.WXN;
    walkparams.uwxn = SCTLR_S.UWXN;
    walkparams.ntlsmid = if HaveTrapLoadStoreMultipleDeviceExt() then SCTLR_S.nTlSMID else '1';
    walkparams.mair = MAIR1_S:MAIR0_S;
    walkparams.sif = SCR.SIF;

    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    if varange == VARange\_LOWER then
        walkparams.sh = TTBCR_S.SH0;
        walkparams.irgn = TTBCR_S.IRGN0;
        walkparams.orgn = TTBCR_S.ORGNO;
        walkparams.hpd = if AArch32.HaveHPDExt() then TTBCR_S.T2E AND TTBCR2_S.HPD0 else '0';
    else
        walkparams.sh = TTBCR_S.SH1;
        walkparams.irgn = TTBCR_S.IRGN1;
        walkparams.orgn = TTBCR_S.ORGNO;
        walkparams.hpd = if AArch32.HaveHPDExt() then TTBCR_S.T2E AND TTBCR2_S.HPD1 else '0';

    return walkparams;
```

## Library pseudocode for aarch64/debug/brbe/BRBCycleCountingEnabled

```
// BRBCycleCountingEnabled()
// =====
// Returns TRUE if the recording of cycle counts is allowed,
// FALSE otherwise.

boolean BRBCycleCountingEnabled()
    if HaveEL\(EL2\) && BRBCR_EL2.CC == '0' then return FALSE;
    if BRBCR_EL1.CC == '0' then return FALSE;
    return TRUE;
```

## Library pseudocode for aarch64/debug/brbe/BRBEBranch

```
// BRBEBranch()
// =====
// Called to write branch record for the following branches when BRB is active:
// direct branches,
// indirect branches,
// direct branches with link,
// indirect branches with link,
// returns from subroutines.

BRBEBranch(BranchType br_type, boolean cond, bits(64) target_address)
    if BranchRecordAllowed(PSTATE.EL) && FilterBranchRecord(br_type, cond) then
        bits(6) branch_type;
        case br_type of
            when BranchType\_DIR
                branch_type = if cond then '001000' else '000000';
            when BranchType\_INDIR
                branch_type = '000001';
            when BranchType\_DIRCALL
                branch_type = '000010';
            when BranchType\_INDCALL
                branch_type = '000011';
            when BranchType\_RET
                branch_type = '000101';
            otherwise
                Unreachable();

        bit ccu;
        bits(14) cc;
        (ccu, cc) = BranchEncCycleCount();
        bit lastfailed = if HaveTME() then BRBFCR_EL1.LASTFAILED else '0';
        bit transactional = if HaveTME() && TSTATE.depth > 0 then '1' else '0';
        bits(2) el = PSTATE.EL;
        bit mispredict = if BRBEMispredictAllowed() && BranchMispredict() then '1' else '0';

        UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional, branch_type, el, mispredict,
                                '11', PC[], target_address);

        BRBFCR_EL1.LASTFAILED = '0';

        PMUEvent(PMU_EVENT_BRB_FILTRATE);

    return;
```

## Library pseudocode for aarch64/debug/brbe/BRBEBranchOnISB

```
// BRBEBranchOnISB()
// =====
// Returns TRUE if ISBs generate Branch records, and FALSE otherwise.

boolean BRBEBranchOnISB()
    return boolean IMPLEMENTATION_DEFINED "ISB generates Branch records";
```

## Library pseudocode for aarch64/debug/brbe/BRBEDebugStateExit

```
// BRBEDebugStateExit()
// =====
// Called to write Debug state exit branch record when BRB is active.

BRBEDebugStateExit(bits(64) target_address)
    if BranchRecordAllowed(PSTATE.EL) then
        // Debug state is a prohibited region, therefore ccu=1, cc=0, source_address=0
        bits(6) branch_type = '111001';
        bit ccu = '1';
        bits(14) cc = Zeros(14);
        bit lastfailed = if HaveTME() then BRBFCCR_EL1.LASTFAILED else '0';
        bit transactional = '0';
        bits(2) el = PSTATE.EL;
        bit mispredict = '0';

        UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional, branch_type, el, mispredict,
                                '01', Zeros(64), target_address);

        BRBFCCR_EL1.LASTFAILED = '0';

        PMUEvent(PMU_EVENT_BRB_FILTRATE);

    return;
```



```

// BRBEEException()
// =====
// Called to write exception branch record when BRB is active.

BRBEEException(ExceptionRecord erec, boolean source_valid,
               bits(64) source_address_in,
               bits(64) target_address_in, bits(2) target_el,
               boolean trappedsyscallinst)
bits(64) target_address = target_address_in;
Exception except = erec.exceptype;
bits(25) iss = erec.syndrome;
case target_el of
  when EL3 if !HaveBRBEv1p1() || (MDCR_EL3.E3BREC == MDCR_EL3.E3BREW) then return;
  when EL2 if BRBCR_EL2.EXCEPTION == '0' then return;
  when EL1 if BRBCR_EL1.EXCEPTION == '0' then return;

boolean target_valid = BranchRecordAllowed(target_el);

if source_valid || target_valid then
  bits(6) branch_type;
  case except of
    when Exception Uncategorized      branch_type = '100011'; // Trap
    when Exception WFXTrap            branch_type = '100011'; // Trap
    when Exception CP15RTTrap         branch_type = '100011'; // Trap
    when Exception CP15RRTTrap        branch_type = '100011'; // Trap
    when Exception CP14RTTrap         branch_type = '100011'; // Trap
    when Exception CP14DTTrap         branch_type = '100011'; // Trap
    when Exception AdvSIMDFPAccessTrap branch_type = '100011'; // Trap
    when Exception FPIDTrap           branch_type = '100011'; // Trap
    when Exception PACTrap            branch_type = '100011'; // Trap
    when Exception TSTARTAccessTrap   branch_type = '100011'; // Trap
    when Exception CP14RRTTrap        branch_type = '100011'; // Trap
    when Exception BranchTarget       branch_type = '101011'; // Inst Fault
    when Exception IllegalState       branch_type = '100011'; // Trap
    when Exception SupervisorCall
      if !trappedsyscallinst then    branch_type = '100010'; // Call
      else                          branch_type = '100011'; // Trap
    when Exception HypervisorCall     branch_type = '100010'; // Call
    when Exception MonitorCall
      if !trappedsyscallinst then    branch_type = '100010'; // Call
      else                          branch_type = '100011'; // Trap
    when Exception SystemRegisterTrap branch_type = '100011'; // Trap
    when Exception SystemRegister128Trap branch_type = '100011'; // Trap
    when Exception SVEAccessTrap      branch_type = '100011'; // Trap
    when Exception SMEAccessTrap      branch_type = '100011'; // Trap
    when Exception ERetTrap           branch_type = '100011'; // Trap
    when Exception PACFail            branch_type = '101100'; // Data Fault
    when Exception InstructionAbort    branch_type = '101011'; // Inst Fault
    when Exception PCAlignment        branch_type = '101010'; // Alignment
    when Exception DataAbort          branch_type = '101100'; // Data Fault
    when Exception NV2DataAbort       branch_type = '101100'; // Data Fault
    when Exception SPAlignment        branch_type = '101010'; // Alignment
    when Exception FPTrappedException branch_type = '100011'; // Trap
    when Exception SError             branch_type = '100100'; // System Error
    when Exception Breakpoint         branch_type = '100110'; // Inst debug
    when Exception SoftwareStep       branch_type = '100110'; // Inst debug
    when Exception Watchpoint         branch_type = '100111'; // Data debug
    when Exception NV2Watchpoint      branch_type = '100111'; // Data debug
    when Exception SoftwareBreakpoint branch_type = '100110'; // Inst debug
    when Exception IRQ                branch_type = '101110'; // IRQ
    when Exception FIQ                branch_type = '101111'; // FIQ
    when Exception MemCpyMemSet       branch_type = '100011'; // Trap
    when Exception GCSFail
      if iss<23:20> == '0000' then    branch_type = '101100'; // Data Fault
      elsif iss<23:20> == '0001' then branch_type = '101011'; // Inst Fault
      elsif iss<23:20> == '0010' then branch_type = '100011'; // Trap
      else                          Unreachable();
    otherwise                        Unreachable();

bit ccu;

```

```

bits(14) cc;
(ccu, cc) = BranchEncCycleCount();
bit lastfailed = if HaveTME() then BRBFCR_EL1.LASTFAILED else '0';
bit transactional = if source_valid && HaveTME() && TSTATE.depth > 0 then '1' else '0';
bits(2) el = if target_valid then target_el else '00';
bit mispredict = '0';
bit sv = if source_valid then '1' else '0';
bit tv = if target_valid then '1' else '0';
bits(64) source_address = if source_valid then source_address_in else Zeros(64);

if !target_valid then
    target_address = Zeros(64);
else
    target_address = AArch64.BranchAddr(target_address, target_el);

UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional,
                           branch_type, el, mispredict,
                           sv:tv, source_address, target_address);

BRBFCR_EL1.LASTFAILED = '0';

PMUEvent(PMU_EVENT_BRB_FILTRATE);

return;

```

### Library pseudocode for aarch64/debug/brbe/BRBExceptionReturn

```

// BRBExceptionReturn()
// =====
// Called to write exception return branch record when BRB is active.

BRBExceptionReturn(bits(64) target_address_in, bits(2) source_el,
                   boolean source_valid, bits(64) source_address_in)
bits(64) target_address = target_address_in;
case source_el of
    when EL3 if !HaveBRBEv1p1() || (MDCR_EL3.E3BREC == MDCR_EL3.E3BREW) then return;
    when EL2 if BRBCR_EL2.ERTN == '0' then return;
    when EL1 if BRBCR_EL1.ERTN == '0' then return;

boolean target_valid = BranchRecordAllowed(PSTATE.EL);

if source_valid || target_valid then
    bits(6) branch_type = '000111';
    bit ccu;
    bits(14) cc;
    (ccu, cc) = BranchEncCycleCount();
    bit lastfailed = if HaveTME() then BRBFCR_EL1.LASTFAILED else '0';
    bit transactional = if source_valid && HaveTME() && TSTATE.depth > 0 then '1' else '0';
    bits(2) el = if target_valid then PSTATE.EL else '00';
    bit mispredict = if (source_valid && BRBEMispredictAllowed() &&
                        BranchMispredict()) then '1' else '0';
    bit sv = if source_valid then '1' else '0';
    bit tv = if target_valid then '1' else '0';
    bits(64) source_address = if source_valid then source_address_in else Zeros(64);
    if !target_valid then
        target_address = Zeros(64);

    UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional,
                             branch_type, el, mispredict,
                             sv:tv, source_address, target_address);

    BRBFCR_EL1.LASTFAILED = '0';

    PMUEvent(PMU_EVENT_BRB_FILTRATE);

return;

```

## Library pseudocode for aarch64/debug/brbe/BRBEFreeze

```
// BRBEFreeze()
// =====
// Generates BRBE freeze event.

BRBEFreeze()
    BRBFCR_EL1.PAUSED = '1';
    BRBTS_EL1 = GetTimeStamp(BRBETimeStamp());
```

## Library pseudocode for aarch64/debug/brbe/BRBEISB

```
// BRBEISB()
// =====
// Handles ISB instruction for BRBE.

BRBEISB()
    boolean branch_conditional = FALSE;
    BRBEBranch(BranchType\_DIR, branch_conditional, PC[] + 4);
```

## Library pseudocode for aarch64/debug/brbe/BRBEMispredictAllowed

```
// BRBEMispredictAllowed()
// =====
// Returns TRUE if the recording of branch misprediction is allowed,
// FALSE otherwise.

boolean BRBEMispredictAllowed()
    if HaveEL(EL2) && BRBCR_EL2.MPRED == '0' then return FALSE;
    if BRBCR_EL1.MPRED == '0' then return FALSE;
    return TRUE;
```

## Library pseudocode for aarch64/debug/brbe/BRBETimeStamp

```
// BRBETimeStamp()
// =====
// Returns captured timestamp.

TimeStamp BRBETimeStamp()
    if HaveEL(EL2) then
        TS_el2 = BRBCR_EL2.TS;
        if !HaveECVExt() && TS_el2 == '10' then
            // Reserved value
            (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable\_EL2TIMESTAMP, 2);
        case TS_el2 of
            when '00'
                // Falls out to check BRBCR_EL1.TS
            when '01'
                return TimeStamp\_Virtual;
            when '10'
                assert HaveECVExt(); // Otherwise ConstrainUnpredictableBits removes this case
                return TimeStamp\_OffsetPhysical;
            when '11'
                return TimeStamp\_Physical;

    TS_el1 = BRBCR_EL1.TS;
    if TS_el1 == '00' || (!HaveECVExt() && TS_el1 == '10') then
        // Reserved value
        (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable\_EL1TIMESTAMP, 2);
    case TS_el1 of
        when '01'
            return TimeStamp\_Virtual;
        when '10'
            return TimeStamp\_OffsetPhysical;
        when '11'
            return TimeStamp\_Physical;
    otherwise
        Unreachable(); // ConstrainUnpredictableBits removes this case
```

## Library pseudocode for aarch64/debug/brbe/BRB\_IALL

```
// BRB_IALL()
// =====
// Called to perform invalidation of branch records

BRB_IALL()
    for i = 0 to GetBRBNumRecords() - 1
        Records_SRC[i] = Zeros(64);
        Records_TGT[i] = Zeros(64);
        Records_INF[i] = Zeros(64);
```

## Library pseudocode for aarch64/debug/brbe/BRB\_INJ

```
// BRB_INJ()
// =====
// Called to perform manual injection of branch records.

BRB_INJ()
    UpdateBranchRecordBuffer(BRBINFINJ_EL1.CCU, BRBINFINJ_EL1.CC, BRBINFINJ_EL1.LASTFAILED,
                               BRBINFINJ_EL1.T, BRBINFINJ_EL1.TYPE, BRBINFINJ_EL1.EL,
                               BRBINFINJ_EL1.MPRED, BRBINFINJ_EL1.VALID, BRBSRCINJ_EL1.ADDRESS,
                               BRBTGTINJ_EL1.ADDRESS);

    BRBINFINJ_EL1 = bits(64) UNKNOWN;
    BRBSRCINJ_EL1 = bits(64) UNKNOWN;
    BRBTGTINJ_EL1 = bits(64) UNKNOWN;

    if ConstrainUnpredictableBool(Unpredictable\_BRBFILTRATE) then PMUEvent(PMU_EVENT_BRB_FILTRATE);
```

## Library pseudocode for aarch64/debug/brbe/Branch

```
type BRBSRCType;
type BRBTGTType;
type BRBINFType;
```

## Library pseudocode for aarch64/debug/brbe/BranchEncCycleCount

```
// BranchEncCycleCount()
// =====
// The first return result is '1' if either of the following is true, and '0' otherwise:
// - This is the first Branch record after the PE exited a Prohibited Region.
// - This is the first Branch record after cycle counting has been enabled.
// If the first return result is '0', the second return result is the encoded cycle count
// since the last branch.
// The format of this field uses a mantissa and exponent to express the cycle count value.
// - bits[7:0] indicate the mantissa M.
// - bits[13:8] indicate the exponent E.
// The cycle count is expressed using the following function:
//   cycle_count = (if IsZero(E) then UInt(M) else UInt('1':M:Zeros(UInt(E)-1)))
// A value of all ones in both the mantissa and exponent indicates the cycle count value
// exceeded the size of the cycle counter.
// If the cycle count is not known, the second return result is zero.

(bit, bits(14)) BranchEncCycleCount();
```

## Library pseudocode for aarch64/debug/brbe/BranchMispredict

```
// BranchMispredict()
// =====
// Returns TRUE if the branch being executed was mispredicted, FALSE otherwise.

boolean BranchMispredict();
```



## Library pseudocode for aarch64/debug/brbe/BranchRawCycleCount

```
// BranchRawCycleCount()
// =====
// If the cycle count is known, the return result is the cycle count since the last branch.

integer BranchRawCycleCount();
```

## Library pseudocode for aarch64/debug/brbe/BranchRecordAllowed

```
// BranchRecordAllowed()
// =====
// Returns TRUE if branch recording is allowed, FALSE otherwise.

boolean BranchRecordAllowed(bits(2) el)
    if ELUsingAArch32(el) then
        return FALSE;

    if BRBFCCR_EL1.PAUSED == '1' then
        return FALSE;

    if el == EL3 && HaveBRBEv1p1() then
        return (MDCR_EL3.E3BREC != MDCR_EL3.E3BREW);

    if HaveEL(EL3) && (MDCR_EL3.SBRBE == '00' ||
        (CurrentSecurityState() == SS\_Secure && MDCR_EL3.SBRBE == '01')) then
        return FALSE;

    case el of
        when EL3 return FALSE; // FEAT_BRBEv1p1 not implemented
        when EL2 return BRBCR_EL2.E2BRE == '1';
        when EL1 return BRBCR_EL1.E1BRE == '1';
        when EL0
            if EL2Enabled() && HCR_EL2.TGE == '1' then
                return BRBCR_EL2.E0HBRE == '1';
            else
                return BRBCR_EL1.E0BRE == '1';
```

## Library pseudocode for aarch64/debug/brbe/Contents

```
// Contents of the Branch Record Buffer
//=====

array [0..63] of BRBSRCType Records_SRC;

array [0..63] of BRBTGTType Records_TGT;

array [0..63] of BRBINFType Records_INF;
```

### Library pseudocode for aarch64/debug/brbe/FilterBranchRecord

```
// FilterBranchRecord()
// =====
// Returns TRUE if the branch record is not filtered out, FALSE otherwise.

boolean FilterBranchRecord(BranchType br, boolean cond)
    case br of
        when BranchType\_DIRCALL
            return BRBFCE_EL1.DIRCALL != BRBFCE_EL1.EnI;
        when BranchType\_INDCALL
            return BRBFCE_EL1.INDCALL != BRBFCE_EL1.EnI;
        when BranchType\_RET
            return BRBFCE_EL1.RTN != BRBFCE_EL1.EnI;
        when BranchType\_DIR
            if cond then
                return BRBFCE_EL1.CONDDIR != BRBFCE_EL1.EnI;
            else
                return BRBFCE_EL1.DIRECT != BRBFCE_EL1.EnI;
        when BranchType\_INDIR
            return BRBFCE_EL1.INDIRECT != BRBFCE_EL1.EnI;
        otherwise Unreachable();
    return FALSE;
```

### Library pseudocode for aarch64/debug/brbe/FirstBranchAfterProhibited

```
// FirstBranchAfterProhibited()
// =====
// Returns TRUE if branch recorded is the first branch after a prohibited region,
// FALSE otherwise.

FirstBranchAfterProhibited();
```

### Library pseudocode for aarch64/debug/brbe/GetBRBNumRecords

```
// GetBRBNumRecords()
// =====
// Returns the number of branch records implemented.

integer GetBRBNumRecords()
    assert UInt(BRBIDR0_EL1.NUMREC) IN {0x08, 0x10, 0x20, 0x40};
    return integer IMPLEMENTATION_DEFINED "Number of BRB records";
```

## Library pseudocode for aarch64/debug/brbe/Getter

```
// Getter functions for branch records
// =====
// Functions used by MRS instructions that access branch records

BRBSRCType BRBSRC_EL1[integer n]
  assert n IN {0..31};
  integer record = UInt(BRBFCR_EL1.BANK:n<4:0>);
  if record < GetBRBNumRecords() then
    return Records_SRC[record];
  else
    return Zeros(64);

BRBTGTType BRBTGT_EL1[integer n]
  assert n IN {0..31};
  integer record = UInt(BRBFCR_EL1.BANK:n<4:0>);
  if record < GetBRBNumRecords() then
    return Records_TGT[record];
  else
    return Zeros(64);

BRBINFType BRBINF_EL1[integer n]
  assert n IN {0..31};
  integer record = UInt(BRBFCR_EL1.BANK:n<4:0>);
  if record < GetBRBNumRecords() then
    return Records_INF[record];
  else
    return Zeros(64);
```

## Library pseudocode for aarch64/debug/brbe/ShouldBRBEFreeze

```
// ShouldBRBEFreeze()
// =====
// Returns TRUE if the BRBE freeze event conditions have been met, and FALSE otherwise.

boolean ShouldBRBEFreeze()
  if !BranchRecordAllowed(PSTATE.EL) then return FALSE;
  boolean check_e      = FALSE;
  boolean check_cnten  = FALSE;
  boolean check_inten  = FALSE;
  boolean exclude_sync = FALSE;
  boolean exclude_cyc  = TRUE;
  boolean include_lo;
  boolean include_hi;

  if HaveEL(EL2) then
    include_lo = (BRBCR_EL1.FZP == '1');
    include_hi = (BRBCR_EL2.FZP == '1');
  else
    include_lo = TRUE;
    include_hi = TRUE;

  return PMUOverflowCondition(check_e, check_cnten, check_inten,
                             include_hi, include_lo, exclude_cyc,
                             exclude_sync);
```

## Library pseudocode for aarch64/debug/brbe/UpdateBranchRecordBuffer

```
// UpdateBranchRecordBuffer()
// =====
// Add a new Branch record to the buffer.

UpdateBranchRecordBuffer(bit ccu, bits(14) cc, bit lastfailed, bit transactional,
                        bits(6) branch_type, bits(2) el, bit mispredict, bits(2) valid,
                        bits(64) source_address, bits(64) target_address)
// Shift the Branch Records in the buffer
for i = GetBRBENumRecords() - 1 downto 1
    Records_SRC[i] = Records_SRC[i - 1];
    Records_TGT[i] = Records_TGT[i - 1];
    Records_INF[i] = Records_INF[i - 1];

Records_INF[0].CCU      = ccu;
Records_INF[0].CC       = cc;

Records_INF[0].EL       = el;
Records_INF[0].VALID    = valid;
Records_INF[0].T        = transactional;
Records_INF[0].LASTFAILED = lastfailed;
Records_INF[0].MPRED     = mispredict;
Records_INF[0].TYPE      = branch_type;

Records_SRC[0] = source_address;
Records_TGT[0] = target_address;

return;
```

## Library pseudocode for aarch64/debug/breakpoint/AArch64.BreakpointMatch

```
// AArch64.BreakpointMatch()
// =====
// Breakpoint matching in an AArch64 translation regime.
// Returns a pair of booleans, the first indicates if the match was successful and the second if
// the first value should be inverted because the breakpoint is configured for a mismatch.

(boolean, boolean) AArch64.BreakpointMatch(integer n, bits(64) vaddress, AccessDescriptor accdesc,
                                             integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n < NumBreakpointsImplemented();

    linking_enabled = (DBGBCR_EL1[n].BT IN {'0x11', '1xx1'} ||
                       (HaveFeatABLE() && DBGBCR_EL1[n].BT2 == '1'));

    // A breakpoint that has linking enabled does not generate debug events in isolation
    if linking_enabled then
        return (FALSE, FALSE);

    enabled      = IsBreakpointEnabled(n);
    linked       = DBGBCR_EL1[n].BT IN {'0x01'};
    isbreakpnt   = TRUE;
    linked_to    = FALSE;
    lbnx         = if Havev8p9Debug() then DBGBCR_EL1[n].LBNX else '00';
    linked_n     = UInt(lbnx : DBGBCR_EL1[n].LBN);
    ssce         = if HaveRME() then DBGBCR_EL1[n].SSCE else '0';
    state_match  = AArch64.StateMatch(DBGBCR_EL1[n].SSC, ssce, DBGBCR_EL1[n].HMC,
                                     DBGBCR_EL1[n].PMC, linked, linked_n, isbreakpnt,
                                     vaddress, accdesc);

    (value_match, valid_mismatch) = AArch64.BreakpointValueMatch(n, vaddress, linked_to,
                                                                isbreakpnt);

    if HaveAArch32() && size == 4 then // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        (match_i, -) = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to,
                                                    isbreakpnt);

        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable\_BPMATCHHALF);

    if vaddress<1> == '1' && DBGBCR_EL1[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR_EL1[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBCR_EL1[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool(Unpredictable\_BPMATCHHALF);

    match = value_match && state_match && enabled;
    is_mismatch = valid_mismatch && state_match && enabled;

    return (match, is_mismatch);
```



```

// AArch64.BreakpointValueMatch()
// =====
// Returns a pair of booleans, the first indicates if the value match was successful and the
// second if the first value should be inverted for a mismatch.

(boolean, boolean) AArch64.BreakpointValueMatch(integer n_in, bits(64) vaddress,
                                                boolean linked_to, boolean isbreakpnt)

// "n" is the identity of the breakpoint unit to match against.
// "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
// matching breakpoints.
// "linked_to" is TRUE if this is a call from StateMatch for linking.
// "isbreakpnt" TRUE is this is a call from BreakpointMatch or from StateMatch for a
// linked breakpoint.
integer n = n_in;
Constraint c;

// If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
// no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
if n >= NumBreakpointsImplemented() then
    (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1,
                                         Unpredictable\_BPNOTIMPL);
    assert c IN {Constraint\_DISABLED, Constraint\_UNKNOWN};
    if c == Constraint\_DISABLED then return (FALSE, FALSE);

// If this breakpoint is not enabled, it cannot generate a match.
// (This could also happen on a call from StateMatch for linking).
if !IsBreakpointEnabled(n) then return (FALSE, FALSE);

// If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
dbgtype = DBGBCR_EL1[n].BT;
bt2 = if HaveFeatABLE() then DBGBCR_EL1[n].BT2 else '0';

(c, bt2, dbgtype) = AArch64.ReservedBreakpointType(n, bt2, dbgtype);
if c == Constraint\_DISABLED then return (FALSE, FALSE);
// Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

// Determine what to compare against.
match_addr    = (dbgtype IN {'0x0x'});
mismatch      = (dbgtype IN {'010x'});
match_vmid    = (dbgtype IN {'10xx'});
match_cid     = (dbgtype IN {'001x'});
match_cid1    = (dbgtype IN {'101x', 'x11x'});
match_cid2    = (dbgtype IN {'11xx'});
linking_enabled = (dbgtype IN {'xx11', '1xx1'} || bt2 == '1');

// If this is a call from StateMatch, return FALSE if the breakpoint is not
// programmed with linking enabled.
if linked_to && !linking_enabled then
    return (FALSE, FALSE);

// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked_to && linking_enabled && !match_addr then
    return (FALSE, FALSE);

// If a linked breakpoint is linked to an address matching breakpoint,
// the behavior is CONSTRAINED UNPREDICTABLE.
if linked_to && match_addr && isbreakpnt then
    if !ConstrainUnpredictableBool(Unpredictable\_BPLINKEDADDRMATCH) then
        return (FALSE, FALSE);

// A breakpoint programmed for address mismatch does not match in AArch32 state.
if mismatch && UsingAArch32() then
    return (FALSE, FALSE);

boolean bvr_match = FALSE;
boolean bxvr_match = FALSE;
integer mask;

if HaveFeatABLE() then

```

```

mask = UInt(DBGBCR_EL1[n].MASK);

// If the mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
if mask IN {1, 2} then
    (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable\_RESBPMASK);
    assert c IN {Constraint\_DISABLED, Constraint\_NONE, Constraint\_UNKNOWN};
    case c of
        when Constraint\_DISABLED return (FALSE, FALSE); // Disabled
        when Constraint\_NONE mask = 0; // No masking
        // Otherwise the value returned by ConstrainUnpredictableBits must
        // be a not-reserved value.

if mask != 0 then
    // When DBGBCR_EL1[n].MASK is a valid nonzero value, the behavior is
    // CONSTRAINED UNPREDICTABLE if any of the following are true:
    // - DBGBCR_EL1[n].<BT2,BT> is programmed for a Context matching breakpoint.
    // - DBGBCR_EL1[n].BAS is not '1111' and AArch32 is supported at EL0.
    if ((match_cid || match_cid1 || match_cid2) ||
        (DBGBCR_EL1[n].BAS != '1111' && HaveAArch32())) then
        if !ConstrainUnpredictableBool(Unpredictable\_BPMASK) then return (FALSE, FALSE);
    else
        // A stand-alone mismatch of a single address is not supported.
        if mismatch then
            return (FALSE, FALSE);

else
    mask = 0;

// Do the comparison.
if match_addr then
    boolean byte_select_match;
    integer byte = UInt(vaddress<1:0>);

    if HaveAArch32() then
        // T32 instructions can be executed at EL0 in an AArch64 translation regime.
        assert byte IN {0,2}; // "vaddress" is halfword aligned
        byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
    else
        assert byte == 0; // "vaddress" is word aligned
        byte_select_match = TRUE; // DBGBCR_EL1[n].BAS<byte> is RES1

    // When FEAT_LVA3 is not implemented, if the DBGBCR_EL1[n].RESS field bits are not a
    // sign extension of the MSB of DBGBCR_EL1[n].VA, it is UNPREDICTABLE whether they
    // appear to be included in the match.
    // If 'vaddress' is outside of the current virtual address space, then the access
    // generates a Translation fault.
    integer top = DebugAddrTop();
    integer bottom = 2;
    if (top < 55 && !IsOnes(DBGBCR_EL1[n]<63:top>) && !IsZero(DBGBCR_EL1[n]<63:top>) &&
        ConstrainUnpredictableBool(Unpredictable\_DBGxVR\_RESS)) then
        top = 63;

    if mask > bottom then
        bvr_match = (vaddress<top:mask> == DBGBCR_EL1[n]<top:mask>) && byte_select_match;

        // If masked bits of DBGBCR_EL1[n] are not zero, the behavior
        // is CONSTRAINED UNPREDICTABLE.
        if bvr_match && !IsZero(DBGBCR_EL1[n]<mask-1:bottom>) then
            bvr_match = ConstrainUnpredictableBool(Unpredictable\_BPMASKEDBITS);
        else
            bvr_match = (vaddress<top:bottom> == DBGBCR_EL1[n]<top:bottom>) && byte_select_match;

elseif match_cid then
    if IsInHost() then
        bvr_match = (CONTEXTIDR_EL2<31:0> == DBGBCR_EL1[n]<31:0>);
    else
        bvr_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1<31:0> == DBGBCR_EL1[n]<31:0>);

elseif match_cid1 then
    bvr_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost()) &&

```



```

CONTEXTIDR_EL1<31:0> == DBGEBR_EL1[n]<31:0>);

if match_vmid then
    bits(16) vmid;
    bits(16) bvr_vmid;

    if !Have16bitVMID() || VTCR_EL2.VS == '0' then
        vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        bvr_vmid = ZeroExtend(DBGEBR_EL1[n]<39:32>, 16);
    else
        vmid = VTTBR_EL2.VMID;
        bvr_vmid = DBGEBR_EL1[n]<47:32>;

    bxvr_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() && vmid == bvr_vmid);

elseif match_cid2 then
    bxvr_match = (PSTATE.EL != EL3 && EL2Enabled() &&
        DBGEBR_EL1[n]<63:32> == CONTEXTIDR_EL2<31:0>);

bvr_match_valid = (match_addr || match_cid || match_cid1);
bxvr_match_valid = (match_vmid || match_cid2);

value_match = (!bxvr_match_valid || bxvr_match) && (!bvr_match_valid || bvr_match);

return (value_match, mismatch);

```

### Library pseudocode for aarch64/debug/breakpoint/AArch64.ReservedBreakpointType

```

// AArch64.ReservedBreakpointType()
// =====
// Checks if the given DBGBCR<n>_EL1.BT2 and DBGBCR<n>_EL1.BT value is reserved and will
// generate Constrained Unpredictable behavior, otherwise returns Constraint_NONE.

(Constraint, bit, bits(4)) AArch64.ReservedBreakpointType(integer n, bit bt2_in ,bits(4) bt_in)
    bit bt2 = bt2_in;
    bits(4) bt = bt_in;
    boolean reserved = FALSE;
    context_aware = n >= (NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented());

    if bt2 == '0' then
        // Context matching
        if !(bt IN {'0x0x'}) && !context_aware then
            reserved = TRUE;

        // EL2 extension
        if bt IN {'1xxx'} && !HaveEL(EL2) then
            reserved = TRUE;

        // Context matching
        if bt IN {'011x','11xx'} && !HaveVirtHostExt() && !HaveV82Debug() then
            reserved = TRUE;

        // Reserved
        if bt IN {'010x'} && !HaveFeatABLE() && !HaveAArch32EL(EL1) then
            reserved = TRUE;
    else
        // Reserved
        if !(bt IN {'0x0x'}) then
            reserved = TRUE;

    if reserved then
        Constraint c;
        (c, <bt2, bt>) = ConstrainUnpredictableBits(Unpredictable RESBPTYPE, 5);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, bit UNKNOWN, bits(4) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, bt2, bt);

```



```

// AArch64.StateMatch()
// =====
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) ssc_in, bit ssce_in, bit hmc_in,
                           bits(2) pxc_in, boolean linked_in, integer linked_n_in,
                           boolean isbreakpnt, bits(64) vaddress, AccessDescriptor accdesc)
if !HaveRME() then assert ssce_in == '0';

// "ssc_in", "ssce_in", "hmc_in", "pxc_in" are the control fields from
// the DBGBCR_EL1[n] or DBGWCR_EL1[n] register.
// "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
// "linked_n_in" is the linked breakpoint number from the DBGBCR_EL1[n] or
// DBGWCR_EL1[n] register.
// "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
// "vaddress" is the program counter for a linked watchpoint or the same value passed to
// AArch64.CheckBreakpoint for a linked breakpoint.
// "accdesc" describes the properties of the access being matched.
bits(2) ssc      = ssc_in;
bit ssce         = ssce_in;
bit hmc          = hmc_in;
bits(2) pxc      = pxc_in;
boolean linked   = linked_in;
integer linked_n = linked_n_in;

// If parameters are set to a reserved type, behaves as either disabled or a defined type
Constraint c;
(c, ssc, ssce, hmc, pxc) = CheckValidStateMatch(ssc, ssce, hmc, pxc, isbreakpnt);
if c == Constraint\_DISABLED then return FALSE;
// Otherwise the hmc, ssc, ssce, pxc values are either valid or the values returned by
// CheckValidStateMatch are valid.

EL3_match = HaveEL(EL3) && hmc == '1' && ssc<0> == '0';
EL2_match = HaveEL(EL2) && ((hmc == '1' && (ssc:pxc != '1000')) || ssc == '11');
EL1_match = pxc<0> == '1';
EL0_match = pxc<1> == '1';

boolean priv_match;
case accdesc.el of
  when EL3   priv_match = EL3_match;
  when EL2   priv_match = EL2_match;
  when EL1   priv_match = EL1_match;
  when EL0   priv_match = EL0_match;

// Security state match
boolean ss_match;
case ssce:ssc of
  when '000' ss_match = hmc == '1' || accdesc.ss != SS\_Root;
  when '001' ss_match = accdesc.ss == SS\_NonSecure;
  when '010' ss_match = (hmc == '1' && accdesc.ss == SS\_Root) || accdesc.ss == SS\_Secure;
  when '011' ss_match = (hmc == '1' && accdesc.ss != SS\_Root) || accdesc.ss == SS\_Secure;
  when '101' ss_match = accdesc.ss == SS\_Realm;

boolean linked_match = FALSE;
boolean is_linked_mismatch = FALSE;

if linked then
  // "linked_n" must be an enabled context-aware breakpoint unit. If it is not context-aware
  // then it is CONSTRAINED UNPREDICTABLE whether this gives no match, gives a match without
  // linking, or linked_n is mapped to some UNKNOWN breakpoint that is context-aware.
  if !IsContextMatchingBreakpoint(linked_n) then
    (first_ctx_cmp, last_ctx_cmp) = ContextMatchingBreakpointRange();
    (c, linked_n) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp,
                                                Unpredictable\_BPNOTCTXCMP);
    assert c IN {Constraint\_DISABLED, Constraint\_NONE, Constraint\_UNKNOWN};

    case c of
      when Constraint\_DISABLED return FALSE; // Disabled
      when Constraint\_NONE   linked = FALSE; // No linking
      // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

```

```

if linked then
    linked_to = TRUE;
    (linked_match, is_linked_mismatch) = AArch64.BreakpointValueMatch(linked_n, vaddress,
                                                                    linked_to, isbreakpnt);

return (priv_match && ss_match && (!linked ||
    (!is_linked_mismatch && linked_match) || (is_linked_mismatch && !linked_match)));

```

### Library pseudocode for aarch64/debug/breakpoint/DebugAddrTop

```

// DebugAddrTop()
// =====
// Returns the value for the top bit used in Breakpoint and Watchpoint address comparisons.

integer DebugAddrTop()
    if Have56BitVAExt() then
        return 55;
    elsif Have52BitVAExt() then
        return 52;
    else
        return 48;

```

### Library pseudocode for aarch64/debug/breakpoint/EffectiveMDSELR\_EL1\_BANK

```

// EffectiveMDSELR_EL1_BANK()
// =====
// Return the effective value of MDSELR_EL1.BANK.

bits(2) EffectiveMDSELR_EL1_BANK()
    // If 16 or fewer breakpoints and 16 or fewer watchpoints are implemented,
    // then the field is RES0.
    integer num_bp = NumBreakpointsImplemented();
    integer num_wp = NumWatchpointsImplemented();
    if num_bp <= 16 && num_wp <= 16 then
        return '00';

    // At EL3, the Effective value of this field is zero if MDCR_EL3.EBWE is 0.
    // At EL2, the Effective value is zero if the Effective value of MDCR_EL2.EBWE is 0.
    // That is, if either MDCR_EL3.EBWE is 0 or MDCR_EL2.EBWE is 0.
    // At EL1, the Effective value is zero if the Effective value of MDSCR_EL2.EMBWE is 0.
    // That is, if any of MDCR_EL3.EBWE, MDCR_EL2.EBWE, or MDSCR_EL1.EMBWE is 0.
    if ((HaveEL(EL3) && MDCR_EL3.EBWE == '0') ||
        (PSTATE.EL != EL3 && EL2Enabled() && MDCR_EL2.EBWE == '0') ||
        (PSTATE.EL == EL1 && MDSCR_EL1.EMBWE == '0')) then
        return '00';

    bits(2) bank = MDSELR_EL1.BANK;

    // Values are reserved depending on the number of breakpoints or watchpoints
    // implemented.
    if ((bank == '11' && num_bp <= 48 && num_wp <= 48) ||
        (bank == '10' && num_bp <= 32 && num_wp <= 32)) then
        // Reserved value
        (-, bank) = ConstrainUnpredictableBits(Unpredictable_RESMDSELR, 2);
        // The value returned by ConstrainUnpredictableBits must be a not-reserved value

    return bank;

```

## Library pseudocode for aarch64/debug/breakpoint/IsBreakpointEnabled

```
// IsBreakpointEnabled()
// =====
// Returns TRUE if the effective value of DBGBCR_EL1[n].E is '1', and FALSE otherwise.

boolean IsBreakpointEnabled(integer n)
    if (n > 15 &&
        ((!HaltOnBreakpointOrWatchpoint() && !SelfHostedExtendedBPWPEEnabled()) ||
         (HaltOnBreakpointOrWatchpoint() && EDSCR2.EHBWE == '0')) then
        return FALSE;

    return DBGBCR_EL1[n].E == '1';
```

## Library pseudocode for aarch64/debug/breakpoint/SelfHostedExtendedBPWPEEnabled

```
// SelfHostedExtendedBPWPEEnabled()
// =====
// Returns TRUE if the extended breakpoints and watchpoints are enabled, and FALSE otherwise
// from a self-hosted debug perspective.

boolean SelfHostedExtendedBPWPEEnabled()
    if NumBreakpointsImplemented() <= 16 && NumWatchpointsImplemented() <= 16 then
        return FALSE;

    if ((HaveEL(EL3) && MDCR_EL3.EBWE == '0') ||
        (EL2Enabled() && MDCR_EL2.EBWE == '0')) then
        return FALSE;

    return MDSCR_EL1.EMBWE == '1';
```

## Library pseudocode for aarch64/debug/ebep/CheckForPMUException

```
// CheckForPMUException()
// =====
// Take a PMU exception if enabled, permitted, and unmasked.

CheckForPMUException()
    boolean enabled;
    bits(2) target_el;
    boolean pmu_exception;
    (enabled, target_el) = PMUExceptionEnabled();
    if !enabled || PMUExceptionMasked() then
        pmu_exception = FALSE;
    elsif IsFeatureImplemented(FEAT_SEBEP) && PSTATE.PPEND == '1' then
        pmu_exception = TRUE;
    else
        boolean check_cnten = FALSE;
        boolean check_e = TRUE;
        boolean check_inten = TRUE;
        boolean include_lo = TRUE;
        boolean include_hi = TRUE;
        boolean exclude_cyc = FALSE;
        boolean exclude_sync = IsFeatureImplemented(FEAT_SEBEP);
        pmu_exception = PMUOverflowCondition(check_e, check_cnten, check_inten,
                                             include_hi, include_lo,
                                             exclude_cyc, exclude_sync);

    if pmu_exception then
        TakePMUException(target_el);
```

## Library pseudocode for aarch64/debug/ebep/ExceptionReturnPPEND

```
// ExceptionReturnPPEND()
// =====
// Sets ShouldSetPPEND to the value to write to PSTATE.PPEND
// on an exception return.
// This function is called before any change in Exception level.

ExceptionReturnPPEND(bits(64) spsr)
    boolean enabled_at_source = FALSE;
    boolean masked_at_source = FALSE;
    if spsr<33> == '1' then // SPSR.PPEND
        (enabled_at_source, -) = PMUExceptionEnabled\(\);
        masked_at_source = PMUExceptionMasked\(\);

    bits(2) target_el;
    if IllegalExceptionReturn(spsr) then
        target_el = PSTATE.EL;
    else
        boolean valid;
        (valid, target_el) = ELFromSPSR(spsr);
        assert valid;

    boolean masked_at_dest = PMUExceptionMasked(target_el, spsr<32>); // SPSR.PM
    if enabled_at_source && masked_at_source && !masked_at_dest then
        PSTATE.PPEND = '1';
        ShouldSetPPEND = FALSE;
        // PSTATE.PPEND will not be changed again by this instruction.

    // If PSTATE.PPEND has not been set by this function, ShouldSetPPEND is
    // unchanged, meaning PSTATE.PPEND might either be set by the current instruction
    // causing a counter overflow, or cleared to zero at the end of instruction.

    return;
```

## Library pseudocode for aarch64/debug/ebep/IsSupportingPMUSynchronousMode

```
// IsSupportingPMUSynchronousMode()
// =====
// Returns TRUE if the event support synchronous mode,
// and FALSE otherwise.

boolean IsSupportingPMUSynchronousMode(bits(16) pmuevent);
```

## Library pseudocode for aarch64/debug/ebep/PMUExceptionEnabled

```
// PMUExceptionEnabled()
// =====
// The first return value is TRUE if the PMU exception is enabled, and FALSE otherwise.
// The second return value is the target Exception level for an enabled PMU exception.

(boolean, bits(2)) PMUExceptionEnabled()

    if !IsFeatureImplemented(FEAT_EBEP) then
        return (FALSE, bits(2) UNKNOWN);

    boolean enabled;
    bits(2) target = bits(2) UNKNOWN;

    if HaveEL(EL3) && MDCR_EL3.PMEE != '01' then
        enabled = MDCR_EL3.PMEE == '11';
        if enabled then target = EL3;

    elsif EL2Enabled() && MDCR_EL2.PMEE != '01' then
        enabled = MDCR_EL2.PMEE == '11';
        if enabled then target = EL2;

    else
        bits(2) pmee_el1 = PMECR_EL1.PMEE;
        if pmee_el1 == '01' then // Reserved value
            Constraint c;
            (c, pmee_el1) = ConstrainUnpredictableBits(Unpredictable_RESPMEE, 2);
            assert c IN {Constraint DISABLED, Constraint UNKNOWN};
            if c == Constraint DISABLED then pmee_el1 = '10';
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value

        enabled = pmee_el1 == '11';
        if enabled then
            target = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;

    return (enabled, target);
```

## Library pseudocode for aarch64/debug/ebep/PMUExceptionMasked

```
// PMUExceptionMasked()
// =====
// Return TRUE if the PMU Exception is masked at the current Exception level,
// FALSE otherwise.

boolean PMUExceptionMasked()
    return PMUExceptionMasked(PSTATE.EL, PSTATE.PM);

// PMUExceptionMasked()
// =====
// Return TRUE if the PMU Exception is masked at the specified Exception level
// and by the value of PSTATE.PM, and FALSE otherwise.

boolean PMUExceptionMasked(bits(2) el, bit pm)
    assert IsFeatureImplemented(FEAT_EBEP);

    (-, target) = PMUExceptionEnabled();

    if Halted() then
        return TRUE;
    elsif UInt(target) < UInt(el) then
        return TRUE;
    elsif el == EL2 && target == EL2 && MDCR_EL2.PMEE != '11' then
        return TRUE;
    elsif target == el && (PMECR_EL1.KPME == '0' || pm == '1') then
        return TRUE;

    return FALSE;
```

## Library pseudocode for aarch64/debug/ebep/PMUInterruptEnabled

```
// PMUInterruptEnabled()
// =====
// Return TRUE if the PMU interrupt request (PMUIRQ) is enabled, FALSE otherwise.

boolean PMUInterruptEnabled()
    if !IsFeatureImplemented(FEAT_EBEP) then
        return TRUE;

    boolean enabled;

    if HaveEL(EL3) && MDCR_EL3.PMEE != '01' then
        enabled = MDCR_EL3.PMEE == '00';

    elsif EL2Enabled() && MDCR_EL2.PMEE != '01' then
        enabled = MDCR_EL2.PMEE == '00';

    else
        bits(2) pmee_el1 = PMECR_EL1.PMEE;
        if pmee_el1 == '01' then // Reserved value
            Constraint c;
            (c, pmee_el1) = ConstrainUnpredictableBits(Unpredictable\_RESPMEE, 2);
            assert c IN {Constraint\_DISABLED, Constraint\_UNKNOWN};
            if c == Constraint\_DISABLED then pmee_el1 = '10';
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value
            enabled = pmee_el1 == '00';

    return enabled;
```

## Library pseudocode for aarch64/debug/ebep/TakePMUException

```
// TakePMUException()
// =====
// Takes a PMU exception.

TakePMUException(bits(2) target_el)
    ExceptionRecord except = ExceptionSyndrome(Exception\_PMU);
    bit synchronous = if IsFeatureImplemented(FEAT_SEBEP) then PSTATE.PPEND else '0';
    except.syndrome = Zeros(24) : synchronous;
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/debug/ebep/inst\_addr\_executed

```
bits(64) inst_addr_executed;
```

## Library pseudocode for aarch64/debug/ebep/sync\_counter\_overflowed

```
boolean sync_counter_overflowed;
```

## Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptions

```
// AArch64.GenerateDebugExceptions()
// =====

boolean AArch64.GenerateDebugExceptions()
    ss = CurrentSecurityState();
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, ss, PSTATE.D);
```



## Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom

```
// AArch64.GenerateDebugExceptionsFrom()
// =====

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from_el, SecurityState from_state, bit mask)

    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    route_to_el2 = (HaveEL(EL2) && (from_state != SS\_Secure || IsSecureEL2Enabled())) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');
    target = (if route_to_el2 then EL2 else EL1);
    boolean enabled;
    if HaveEL(EL3) && from_state == SS\_Secure then
        enabled = MDCR_EL3.SDD == '0';
        if from_el == ELO && ELUsingAArch32(EL1) then
            enabled = enabled || SDER32_EL3.SUIDEN == '1';
    else
        enabled = TRUE;

    if from_el == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from_el);

    return enabled;
```

## Library pseudocode for aarch64/debug/ite/AArch64.TRCIT

```
// AArch64.TRCIT()
// =====
// Determines whether an Instrumentation trace packet should
// be generated and then generates an instrumentation trace packet
// containing the value of the register passed as an argument

AArch64.TRCIT(bits(64) Xt)
    ss = CurrentSecurityState();
    if TraceInstrumentationAllowed(ss, PSTATE.EL) then
        TraceInstrumentation(Xt);
```

## Library pseudocode for aarch64/debug/ite/TraceInstrumentation

```
// TraceInstrumentation()
// =====
// Generates an instrumentation trace packet
// containing the value of the register passed as an argument

TraceInstrumentation(bits(64) Xt);
```

## Library pseudocode for aarch64/debug/pmu/AArch64.ClearEventCounters

```
// AArch64.ClearEventCounters()
// =====
// Zero all the event counters.

AArch64.ClearEventCounters()
    integer counters = AArch64.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            PMEVCNTR_ELO[idx] = Zeros(64);
```

## Library pseudocode for aarch64/debug/pmu/AArch64.GetNumEventCountersAccessible

```
// AArch64.GetNumEventCountersAccessible()
// =====
// Return the number of event counters that can be accessed at the current Exception level.

integer AArch64.GetNumEventCountersAccessible()
    integer n;
    integer total_counters = GetNumEventCounters\(\);
    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled\(\) then
        n = UInt(MDCR_EL2.HPMN);
        if n > total_counters || (!HaveFeatHPMNO\(\) && n == 0) then
            (-, n) = ConstrainUnpredictableInteger(0, total_counters,
                Unpredictable\_PMUEVENTCOUNTER);
    else
        n = total_counters;

    return n;
```

## Library pseudocode for aarch64/debug/pmu/AArch64.IncrementCycleCounter

```
// AArch64.IncrementCycleCounter()
// =====
// Increment the cycle counter and possibly set overflow bits.

AArch64.IncrementCycleCounter()
    if !CountPMUEvents(CYCLE\_COUNTER\_ID) then return;
    bit d = PMCR_EL0.D; // Check divide-by-64
    bit lc = PMCR_EL0.LC;
    boolean lc_enabled;
    (lc_enabled, -) = PMUExceptionEnabled();
    lc = if lc_enabled then '1' else lc;
    // Effective value of 'D' bit is 0 when Effective value of LC is '1'
    if lc == '1' then d = '0';
    if d == '1' && !HasElapsed64Cycles() then return;

    integer old_value = UInt(PMCCNTR_EL0);
    integer new_value = old_value + 1;
    PMCCNTR_EL0 = new_value<63:0>;

    integer ovflw;
    if HaveAArch32() then
        ovflw = if lc == '1' then 64 else 32;
    else
        ovflw = 64;

    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET_EL0.C = '1';
        PMOVSLR_EL0.C = '1';

    return;
```

## Library pseudocode for aarch64/debug/pmu/AArch64.IncrementEventCounter

```
// AArch64.IncrementEventCounter()
// =====
// Increment the specified event counter by the specified amount.

AArch64.IncrementEventCounter(integer idx, integer increment)
    integer old_value;
    integer new_value;
    integer ovflw;

    old_value = UInt(PMEVCNTR_EL0[idx]);
    new_value = old_value + PMUCountValue(idx, increment);

    bit lp;
    if HavePMUv3p5() then
        PMEVCNTR_EL0[idx] = new_value<63:0>;
        boolean pmuexception_enabled;
        (pmuexception_enabled, -) = PMUExceptionEnabled();
        if pmuexception_enabled then
            lp = '1';
        else
            lp = if PMUCounterIsHyp(idx) then MDCR_EL2.HLP else PMCR_EL0.LP;
            ovflw = if lp == '1' then 64 else 32;
    else
        PMEVCNTR_EL0[idx] = ZeroExtend(new_value<31:0>, 64);
        ovflw = 32;

    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET_EL0[idx] = '1';
        PMOVSLR_EL0[idx] = '1';
        // Check for the CHAIN event from an even counter
        if idx<0> == '0' && idx + 1 < GetNumEventCounters() && (!HavePMUv3p5() || lp == '0') then
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);
    if (IsFeatureImplemented(FEAT_SEBEP) &&
        IsSupportingPMUSynchronousMode(PMEVTYPEPER_EL0[idx].evtCount) &&
        PMINTENSET_EL1[idx] == '1' && PMOVSLR_EL0[idx] == '1' && increment != 0) then
        SyncCounterOverflowed = TRUE;

    return;
```

## Library pseudocode for aarch64/debug/pmu/AArch64.PMUCycle

```
// AArch64.PMUCycle()
// =====
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

AArch64.PMUCycle()
    if !HavePMUv3() then
        return;

    PMUEvent(PMU_EVENT_CPU_CYCLES);

    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            if ((!IsFeatureImplemented(FEAT_SEBEP) || PMEVTYPER_EL0[idx].SYNC == '0' ||
                !IsSupportingPMUSynchronousMode(PMEVTYPER_EL0[idx].evtCount)) &&
                CountPMUEvents(idx)) then
                integer accumulated = PMUEventAccumulator[idx];
                AArch64.IncrementEventCounter(idx, accumulated);
                PMUEventAccumulator[idx] = 0;
            AArch64.IncrementCycleCounter();
            CheckForPMUOverflow();
```

## Library pseudocode for aarch64/debug/pmu/AArch64.PMUSwIncrement

```
// AArch64.PMUSwIncrement()
// =====
// Generate PMU Events on a write to PMSWINC_EL0.

AArch64.PMUSwIncrement(bits(32) sw_incr)
    integer counters = AArch64.GetNumEventCountersAccessible\(\);
    if counters != 0 then
        for idx = 0 to counters - 1
            if sw_incr<idx> == '1' then
                PMUEvent(PMU_EVENT_SW_INCR, 1, idx);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR1

```
// CollectContextIDR1()
// =====

boolean CollectContextIDR1()
    if !StatisticalProfilingEnabled() then return FALSE;
    if PSTATE.EL == EL2 then return FALSE;
    if EL2Enabled() && HCR_EL2.TGE == '1' then return FALSE;
    return PMSCR_EL1.CX == '1';
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR2

```
// CollectContextIDR2()
// =====

boolean CollectContextIDR2()
    if !StatisticalProfilingEnabled() then return FALSE;
    if !EL2Enabled() then return FALSE;
    return PMSCR_EL2.CX == '1';
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CollectPhysicalAddress

```
// CollectPhysicalAddress()
// =====

boolean CollectPhysicalAddress()
    if !StatisticalProfilingEnabled() then return FALSE;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    if HaveEL(EL2) && (owning_ss != SS\_Secure || IsSecureEL2Enabled()) then
        return PMSCR_EL2.PA == '1' && (owning_el == EL2 || PMSCR_EL1.PA == '1');
    else
        return PMSCR_EL1.PA == '1';
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CollectTimeStamp

```
// CollectTimeStamp()
// =====

TimeStamp CollectTimeStamp()
    if !StatisticalProfilingEnabled\(\) then return TimeStamp\_None;
    (-, owning_el) = ProfilingBufferOwner();

    if owning_el == EL2 then
        if PMSCR_EL2.TS == '0' then return TimeStamp\_None;
    else
        if PMSCR_EL1.TS == '0' then return TimeStamp\_None;

    bits(2) PCT_el1;
    if !HaveECVExt() then
        PCT_el1 = '0':PMSCR_EL1.PCT<0>;          // PCT<1> is RES0
    else
        PCT_el1 = PMSCR_EL1.PCT;
        if PCT_el1 == '10' then
            // Reserved value
            (-, PCT_el1) = ConstrainUnpredictableBits(Unpredictable\_PMSCR\_PCT, 2);
    if EL2Enabled() then
        bits(2) PCT_el2;
        if !HaveECVExt() then
            PCT_el2 = '0':PMSCR_EL2.PCT<0>;      // PCT<1> is RES0
        else
            PCT_el2 = PMSCR_EL2.PCT;
            if PCT_el2 == '10' then
                // Reserved value
                (-, PCT_el2) = ConstrainUnpredictableBits(Unpredictable\_PMSCR\_PCT, 2);
    case PCT_el2 of
        when '00'
            return if IsInHost() then TimeStamp\_Physical else TimeStamp\_Virtual;
        when '01'
            if owning_el == EL2 then return TimeStamp\_Physical;
        when '11'
            assert HaveECVExt();                  // FEAT_ECV must be implemented
            if owning_el == EL1 && PCT_el1 == '00' then
                return if IsInHost() then TimeStamp\_Physical else TimeStamp\_Virtual;
            else
                return TimeStamp\_OffsetPhysical;
        otherwise
            Unreachable();

    case PCT_el1 of
        when '00' return if IsInHost() then TimeStamp\_Physical else TimeStamp\_Virtual;
        when '01' return TimeStamp\_Physical;
        when '11'
            assert HaveECVExt();                  // FEAT_ECV must be implemented
            return TimeStamp\_OffsetPhysical;
        otherwise Unreachable();
```

## Library pseudocode for aarch64/debug/statisticalprofiling/OpType

```
// OpType
// =====
// Types of operation filtered by SPECollectRecord().

enumeration OpType {
    OpType_Load,          // Any memory-read operation other than atomics, compare-and-swap, and swap
    OpType_Store,         // Any memory-write operation, including atomics without return
    OpType_LoadAtomic,    // Atomics with return, compare-and-swap and swap
    OpType_Branch,        // Software write to the PC
    OpType_Other          // Any other class of operation
};
```

## Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferEnabled

```
// ProfilingBufferEnabled()
// =====

boolean ProfilingBufferEnabled()
    if !HaveStatisticalProfiling() then return FALSE;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    bits(2) state_bits;
    if HaveRME() then
        state_bits = SCR_EL3.NSE : EffectiveSCR_EL3_NS();
    else
        state_bits = '0' : SCR_EL3.NS;

    boolean state_match;
    case owning_ss of
        when SS_Secure      state_match = state_bits == '00';
        when SS_NonSecure   state_match = state_bits == '01';
        when SS_Realm       state_match = state_bits == '11';
    return (!ELUsingAArch32(owning_el) && state_match &&
        PMBLIMITR_EL1.E == '1' && PMBSR_EL1.S == '0');
```

## Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferOwner

```
// ProfilingBufferOwner()
// =====

(SecurityState, bits(2)) ProfilingBufferOwner()
    SecurityState owning_ss;

    if HaveEL(EL3) then
        bits(3) state_bits;
        if HaveRME() then
            state_bits = MDCR_EL3.<NSPBE,NSPB>;
            if (state_bits IN {'10x'} ||
                (!HaveSecureEL2Ext() && state_bits IN {'00x'})) then
                // Reserved value
                (-, state_bits) = ConstrainUnpredictableBits(Unpredictable_RESERVEDNSxB, 3);
        else
            state_bits = '0' : MDCR_EL3.NSPB;

        case state_bits of
            when '00x' owning_ss = SS_Secure;
            when '01x' owning_ss = SS_NonSecure;
            when '11x' owning_ss = SS_Realm;
    else
        owning_ss = if SecureOnlyImplementation() then SS_Secure else SS_NonSecure;

    bits(2) owning_el;
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        owning_el = if MDCR_EL2.E2PB == '00' then EL2 else EL1;
    else
        owning_el = EL1;

    return (owning_ss, owning_el);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingSynchronizationBarrier

```
// ProfilingSynchronizationBarrier()
// =====
// Barrier to ensure that all existing profiling data has been formatted, and profiling buffer
// addresses have been translated such that writes to the profiling buffer have been initiated.
// A following DSB completes when writes to the profiling buffer have completed.

ProfilingSynchronizationBarrier();
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEAddByteToRecord

```
// SPEAddByteToRecord()
// =====
// Add one byte to a record and increase size property appropriately.

SPEAddByteToRecord(bits(8) b)
    assert SPERecordSize < SPEMaxRecordSize;
    SPERecordData[SPERecordSize] = b;
    SPERecordSize = SPERecordSize + 1;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEAddPacketToRecord

```
// SPEAddPacketToRecord()
// =====
// Add passed header and payload data to the record.
// Payload must be a multiple of 8.

SPEAddPacketToRecord(bits(2) header_hi, bits(4) header_lo,
                    bits(N) payload)
    assert N MOD 8 == 0;
    bits(2) sz;
    case N of
        when 8 sz = '00';
        when 16 sz = '01';
        when 32 sz = '10';
        when 64 sz = '11';
        otherwise Unreachable();

    bits(8) header = header_hi:sz:header_lo;
    SPEAddByteToRecord(header);
    for i = 0 to (N DIV 8)-1
        SPEAddByteToRecord(payload<i*8+7:i*8>);
```





```

// SPEBranch()
// =====
// Called on every branch if SPE is present. Maintains previous branch target
// and branch related SPE functionality.

SPEBranch(bits(N) target, BranchType branch_type, boolean conditional, boolean taken_flag)
    boolean is_isb = FALSE;
    SPEBranch(target, branch_type, conditional, taken_flag, is_isb);

SPEBranch(bits(N) target, BranchType branch_type, boolean conditional, boolean taken_flag,
    boolean is_isb)
    // If the PE implements branch prediction, data about (mis)prediction is collected
    // through the PMU events.

    boolean collect_prev_br;
    boolean collect_prev_br_eret = boolean IMPLEMENTATION_DEFINED "SPE prev br on eret";
    boolean collect_prev_br_exception = boolean IMPLEMENTATION_DEFINED "SPE prev br on exception";
    boolean collect_prev_br_isb = boolean IMPLEMENTATION_DEFINED "SPE prev br on isb";
    case branch_type of
        when BranchType\_EXCEPTION
            collect_prev_br = collect_prev_br_exception;
        when BranchType\_ERET
            collect_prev_br = collect_prev_br_eret;
        otherwise
            collect_prev_br = !is_isb || collect_prev_br_isb;

    // Implements previous branch target functionality
    if (taken_flag && !IsZero(PMSIDR_EL1.PBT) && StatisticalProfilingEnabled() &&
        collect_prev_br) then

        if SPESampleInFlight then
            // Save the target address for it to be added to record.
            bits(64) previous_target = SPESamplePreviousBranchAddress;
            SPESampleAddress[SPEAddrPosPrevBranchTarget] <63:0> = previous_target <63:0>;
            boolean previous_branch_valid = SPESamplePreviousBranchAddressValid;
            SPESampleAddressValid[SPEAddrPosPrevBranchTarget] = previous_branch_valid;
            SPESamplePreviousBranchAddress <55:0> = target <55:0>;

        bit ns;
        bit nse;
        case CurrentSecurityState() of
            when SS\_Secure
                ns = '0';
                nse = '0';
            when SS\_NonSecure
                ns = '1';
                nse = '0';
            when SS\_Realm
                ns = '1';
                nse = '1';
            otherwise Unreachable();

        SPESamplePreviousBranchAddress <63> = ns;
        SPESamplePreviousBranchAddress <60> = nse;
        SPESamplePreviousBranchAddress <62:61> = PSTATE.EL;
        SPESamplePreviousBranchAddressValid = TRUE;

    if !StatisticalProfilingEnabled() then
        if taken_flag then
            // Invalidate previous branch address, if profiling is disabled
            // or prohibited.
            SPESamplePreviousBranchAddressValid = FALSE;
        return;

    if SPESampleInFlight then
        is_direct = branch_type IN {BranchType\_DIR, BranchType\_DIRCALL};
        SPESampleClass = '10';
        SPESampleSubclass <1> = if is_direct then '0' else '1';
        SPESampleSubclass <0> = if conditional then '1' else '0';
        SPESampleOpType = OpType\_Branch;

```

```

// Save the target address.
if taken_flag then
    SPESampleAddress[SPEAddrPosBranchTarget]<55:0> = target<55:0>;

bit ns;
bit nse;
case CurrentSecurityState() of
    when SS_Secure
        ns = '0';
        nse = '0';
    when SS_NonSecure
        ns = '1';
        nse = '0';
    when SS_Realm
        ns = '1';
        nse = '1';
    otherwise Unreachable();

SPESampleAddress[SPEAddrPosBranchTarget]<63> = ns;
SPESampleAddress[SPEAddrPosBranchTarget]<60> = nse;
SPESampleAddress[SPEAddrPosBranchTarget]<62:61> = PSTATE.EL;
SPESampleAddressValid[SPEAddrPosBranchTarget] = TRUE;

SPESampleEvents<6> = if !taken_flag then '1' else '0';

```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEBufferFilled

```

// SPEBufferFilled()
// =====
// Deal with a full buffer event.

SPEBufferFilled()
    if IsZero(PMBSR_EL1.S) then
        PMBSR_EL1.S = '1'; // Assert PMBIRQ
        PMBSR_EL1.EC = '000000'; // Other buffer management event
        PMBSR_EL1.MSS = ZeroExtend('000001', 16); // Set buffer full event

    PMUEvent(PMU_EVENT_SAMPLE_WRAP);

```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEBufferIsFull

```

// SPEBufferIsFull()
// =====
// Return true if another full size sample record would not fit in the
// profiling buffer.

boolean SPEBufferIsFull()
    integer write_pointer_limit = UInt(PMBLIMTR_EL1.LIMIT:Zeros(12));
    integer current_write_pointer = UInt(PMBPTR_EL1);
    integer record_max_size = 1<<UInt(PMSIDR_EL1.MaxSize);
    return current_write_pointer > (write_pointer_limit - record_max_size);

```



```

// SPECCollectRecord()
// =====
// Returns TRUE if the sampled class of instructions or operations, as
// determined by PMSFCR_EL1, are recorded and FALSE otherwise.

boolean SPECCollectRecord(bits(64) events, integer total_latency, OpType optype)
    assert StatisticalProfilingEnabled();

    bits(64) mask = 0xAA<63:0>; // Bits [7,5,3,1]
    bits(64) e;
    bits(64) m;
    if HaveSVE() then mask<18:17> = '11'; // Predicate flags
    if HaveTME() then mask<16> = '1';
    if HaveStatisticalProfilingv1p1() then mask<11> = '1'; // Alignment Flag
    if HaveStatisticalProfilingv1p2() then mask<6> = '1'; // Not taken flag
    if HaveStatisticalProfilingv1p4() then
        mask<10:8,4,2> = '11111';
    else
        bits(5) impdef_mask;
        impdef_mask = bits(5) IMPLEMENTATION_DEFINED "SPE mask 10:8,4,2";
        mask<10:8,4,2> = impdef_mask;

    mask<63:48> = bits(16) IMPLEMENTATION_DEFINED "SPE mask 63:48";
    mask<31:24> = bits(8) IMPLEMENTATION_DEFINED "SPE mask 31:24";
    mask<15:12> = bits(4) IMPLEMENTATION_DEFINED "SPE mask 15:12";

    e = events AND mask;
    boolean is_rejected_nevent = FALSE;
    boolean is_nevt;
    // Filtering by inverse event
    if HaveStatisticalProfilingv1p2() then
        m = PMSNEVFR_EL1 AND mask;
        is_nevt = IsZero(e AND m);
        if PMSFCR_EL1.FnE == '1' then
            // Inverse filtering by event is enabled
            if !IsZero(m) then
                // Not UNPREDICTABLE case
                is_rejected_nevent = !is_nevt;
            else
                is_rejected_nevent = ConstrainUnpredictableBool(Unpredictable\_BADPMSFCR);
        else
            is_nevt = TRUE; // not implemented

    boolean is_rejected_event = FALSE;

    // Filtering by event
    m = PMSEVFR_EL1 AND mask;
    boolean is_evt = IsZero(NOT(e) AND m);
    if PMSFCR_EL1.FE == '1' then
        // Filtering by event is enabled
        if !IsZero(m) then
            // Not UNPREDICTABLE case
            is_rejected_event = !is_evt;
        else
            is_rejected_event = ConstrainUnpredictableBool(Unpredictable\_BADPMSFCR);

    if (HaveStatisticalProfilingv1p2() && PMSFCR_EL1.<FnE,FE> == '11' &&
        !IsZero(PMSEVFR_EL1 AND PMSNEVFR_EL1 AND mask)) then
        // UNPREDICTABLE case due to combination of filter and inverse filter
        is_rejected_nevent = ConstrainUnpredictableBool(Unpredictable\_BADPMSFCR);
        is_rejected_event = ConstrainUnpredictableBool(Unpredictable\_BADPMSFCR);

    if is_evt && is_nevt then
        PMUEvent(PMU\_EVENT\_SAMPLE\_FEED\_EVENT);

    boolean is_op_br = FALSE;
    boolean is_op_ld = FALSE;
    boolean is_op_st = FALSE;

    is_op_br = (optype == OpType\_Branch);

```

```

is_op_ld = (optype IN {OpType_Load, OpType_LoadAtomic});
is_op_st = (optype IN {OpType_Store, OpType_LoadAtomic});

if is_op_br then PMUEvent(PMU_EVENT_SAMPLE_FEED_BR);
if is_op_ld then PMUEvent(PMU_EVENT_SAMPLE_FEED_LD);
if is_op_st then PMUEvent(PMU_EVENT_SAMPLE_FEED_ST);

boolean is_op = ((is_op_br && PMSFCR_EL1.B == '1') ||
                 (is_op_ld && PMSFCR_EL1.LD == '1') ||
                 (is_op_st && PMSFCR_EL1.ST == '1'));

if is_op then PMUEvent(PMU_EVENT_SAMPLE_FEED_OP);

// Filter by type
boolean is_rejected_type = FALSE;
if PMSFCR_EL1.FT == '1' then
    // Filtering by type is enabled
    if !IsZero(PMSFCR_EL1.<B, LD, ST>) then
        // Not an UNPREDICTABLE case
        is_rejected_type = !is_op;
    else
        is_rejected_type = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);

// Filter by latency
boolean is_rejected_latency = FALSE;
boolean is_lat = (total_latency < UInt(PMSLATFR_EL1.MINLAT));
if is_lat then PMUEvent(PMU_EVENT_SAMPLE_FEED_LAT);

if PMSFCR_EL1.FL == '1' then
    // Filtering by latency is enabled
    if !IsZero(PMSLATFR_EL1.MINLAT) then
        // Not an UNPREDICTABLE case
        is_rejected_latency = !is_lat;
    else
        is_rejected_latency = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);

boolean is_rejected_data_source;
// Filtering by Data Source
if (HaveStatisticalProfilingFDS() && PMSFCR_EL1.FDS == '1' &&
    is_op_ld && SPESampleDataSourceValid) then
    bits(16) data_source = SPESampleDataSource;
    integer index = UInt(data_source<5:0>);
    is_rejected_data_source = PMSDSFR_EL1<index> == '0';
else
    is_rejected_data_source = FALSE;

boolean return_value;
return_value = !(is_rejected_nevent || is_rejected_event ||
                 is_rejected_type || is_rejected_latency);

if return_value then
    PMUEvent(PMU_EVENT_SAMPLE_FILTRATE);
return return_value;

```



```

// SPEConstructRecord()
// =====
// Create new record and populate it with packets using sample storage data.
// This is an example implementation, packets may appear in
// any order as long as the record ends with an End or Timestamp packet.

SPEConstructRecord()
    integer payload_size;

    // Empty the record.
    SPEEmptyRecord();

    // Add contextEL1 if available
    if SPESampleContextEL1Valid then
        SPEAddPacketToRecord('01', '0100', SPESampleContextEL1);

    // Add contextEL2 if available
    if SPESampleContextEL2Valid then
        SPEAddPacketToRecord('01', '0101', SPESampleContextEL2);

    // Add valid counters
    for counter_index = 0 to (SPEMaxCounters - 1)
        if SPESampleCounterValid[counter_index] then
            if counter_index >= 8 then
                // Need extended format
                SPEAddByteToRecord('001000':counter_index<4:3>);
            // Check for overflow
            boolean large_counters = boolean IMPLEMENTATION_DEFINED "SPE 16bit counters";
            if SPESampleCounter[counter_index] > 0xFFFF && large_counters then
                SPESampleCounter[counter_index] = 0xFFFF;
            elsif SPESampleCounter[counter_index] > 0xFFFF then
                SPESampleCounter[counter_index] = 0xFFFF;

            // Add byte0 for short format (byte1 for extended format)
            SPEAddPacketToRecord('10', '1':counter_index<2:0>,
                SPESampleCounter[counter_index]<15:0>);

    // Add valid addresses
    if HaveStatisticalProfilingvlp2() then
        // Under the some conditions, it is IMPLEMENTATION_DEFINED whether
        // previous branch packet is present.
        boolean include_prev_br = boolean IMPLEMENTATION_DEFINED "SPE get prev br if not br";
        if SPESampleOpType != OpType_Branch && !include_prev_br then
            SPESampleAddressValid[SPEAddrPosPrevBranchTarget] = FALSE;

    // Data Virtual address should not be collected if this was an NV2 access and Statistical
    // Profiling is disabled at EL2.
    if !StatisticalProfilingEnabled(EL2) && SPESampleInstIsNV2 then
        SPESampleAddressValid[SPEAddrPosDataVirtual] = FALSE;

    for address_index = 0 to (SPEMaxAddrs - 1)
        if SPESampleAddressValid[address_index] then
            if address_index >= 8 then
                // Need extended format
                SPEAddByteToRecord('001000':address_index<4:3>);
            // Add byte0 for short format (byte1 for extended format)
            SPEAddPacketToRecord('10', '0':address_index<2:0>,
                SPESampleAddress[address_index]);

    // Add Data Source
    if SPESampleDataSourceValid then
        payload_size = SPEGetDataSourcePayloadSize();
        SPEAddPacketToRecord('01', '0011', SPESampleDataSource<8*payload_size-1:0>);

    // Add operation details
    SPEAddPacketToRecord('01', '10':SPESampleClass, SPESampleSubclass);

    // Add events
    // Get size of payload in bytes.
    payload_size = SPEGetEventsPayloadSize();

```

```

SPEAddPacketToRecord('01', '0010', SPESampleEvents<8*payload_size-1:0>);

// Add Timestamp to end the record if one is available.
// Otherwise end with an End packet.
if SPESampleTimestampValid then
    SPEAddPacketToRecord('01', '0001', SPESampleTimestamp);
else
    SPEAddByteToRecord('00000001');

// Add padding
while SPERecordSize MOD (1<<UInt(PMBIDR_EL1.Align)) != 0 do
    SPEAddByteToRecord(Zeros(8));
SPEWriteToBuffer();

```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPECycle

```

// SPECycle()
// =====
// Function called at the end of every cycle. Responsible for asserting interrupts
// and advancing counters.

SPECycle()
    if !HaveStatisticalProfiling() then
        return;

    // Increment pending counters
    if SPESampleInFlight then
        for i = 0 to (SPEMaxCounters - 1)
            if SPESampleCounterPending[i] then
                SPESampleCounter[i] = SPESampleCounter[i] + 1;

    // Assert PMBIRQ if appropriate.
    SetInterruptRequestLevel(InterruptID\_PMBIRQ,
        if PMBSR_EL1.S == '1' then Signal\_High else Signal\_Low);

```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEEmptyRecord

```

// SPEEmptyRecord()
// =====
// Reset record data.

SPEEmptyRecord()
    SPERecordSize = 0;
    for i = 0 to (SPEMaxRecordSize - 1)
        SPERecordData[i] = Zeros(8);

```



## Library pseudocode for aarch64/debug/statisticalprofiling/SPEEvent

```
// SPEEvent()
// =====
// Called by PMUEvent if a sample is in flight.
// Sets appropriate bit in SPESampleStorage.events.

SPEEvent(bits(16) pmuevent)
  case pmuevent of
    when PMU\_EVENT\_DSNP\_HIT\_RD
      if HaveStatisticalProfilingv1p4() then
        SPESampleEvents<23> = '1';
    when PMU\_EVENT\_L1D\_LFB\_HIT\_RD
      if HaveStatisticalProfilingv1p4() then
        SPESampleEvents<22> = '1';
    when PMU\_EVENT\_L2D\_LFB\_HIT\_RD
      if HaveStatisticalProfilingv1p4() then
        SPESampleEvents<22> = '1';
    when PMU\_EVENT\_L3D\_LFB\_HIT\_RD
      if HaveStatisticalProfilingv1p4() then
        SPESampleEvents<22> = '1';
    when PMU\_EVENT\_LL\_LFB\_HIT\_RD
      if HaveStatisticalProfilingv1p4() then
        SPESampleEvents<22> = '1';
    when PMU\_EVENT\_L1D\_CACHE\_HITM\_RD
      if HaveStatisticalProfilingv1p4() then
        SPESampleEvents<21> = '1';
    when PMU\_EVENT\_L2D\_CACHE\_HITM\_RD
      if HaveStatisticalProfilingv1p4() then
        SPESampleEvents<21> = '1';
    when PMU\_EVENT\_L3D\_CACHE\_HITM\_RD
      if HaveStatisticalProfilingv1p4() then
        SPESampleEvents<21> = '1';
    when PMU\_EVENT\_LL\_CACHE\_HITM\_RD
      if HaveStatisticalProfilingv1p4() then
        SPESampleEvents<21> = '1';
    when PMU\_EVENT\_L2D\_CACHE\_LMISS\_RD
      if HaveStatisticalProfilingv1p4() then
        SPESampleEvents<20> = '1';
    when PMU\_EVENT\_L2D\_CACHE\_RD
      if HaveStatisticalProfilingv1p4() then
        SPESampleEvents<19> = '1';
    when PMU\_EVENT\_SVE\_PRED\_EMPTY\_SPEC
      if HaveStatisticalProfilingv1p1() then
        SPESampleEvents<18> = '1';
    when PMU\_EVENT\_SVE\_PRED\_PARTIAL\_SPEC
      if HaveStatisticalProfilingv1p1() then
        SPESampleEvents<17> = '1';
    when PMU\_EVENT\_LDST\_ALIGN\_LAT
      if HaveStatisticalProfilingv1p1() then
        SPESampleEvents<11> = '1';
    when PMU\_EVENT\_REMOTE\_ACCESS
      SPESampleEvents<10> = '1';
    when PMU\_EVENT\_LL\_CACHE\_MISS
      SPESampleEvents<9> = '1';
    when PMU\_EVENT\_LL\_CACHE
      SPESampleEvents<8> = '1';
    when PMU\_EVENT\_BR\_MIS\_PRED
      SPESampleEvents<7> = '1';
    when PMU\_EVENT\_BR\_MIS\_PRED\_RETIRE
      SPESampleEvents<7> = '1';
    when PMU\_EVENT\_DTLB\_WALK
      SPESampleEvents<5> = '1';
    when PMU\_EVENT\_L1D\_TLB
      SPESampleEvents<4> = '1';
    when PMU\_EVENT\_L1D\_CACHE\_REFILL
      if !HaveStatisticalProfilingv1p4() then
        SPESampleEvents<3> = '1';
    when PMU\_EVENT\_L1D\_CACHE\_LMISS\_RD
      if HaveStatisticalProfilingv1p4() then
        SPESampleEvents<3> = '1';
    when PMU\_EVENT\_L1D\_CACHE
      SPESampleEvents<2> = '1';
    when PMU\_EVENT\_INST\_RETIRE
      SPESampleEvents<1> = '1';
    when PMU\_EVENT\_EXC\_TAKEN
      SPESampleEvents<0> = '1';
    otherwise return;
  return;
```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEFreezeOnEvent

```
// SPEFreezeOnEvent()
// =====
// Returns TRUE if PMU event counter idx should be frozen due to an SPE event, and FALSE otherwise.

boolean SPEFreezeOnEvent(integer idx)
    assert 0 <= idx;
    if !IsFeatureImplemented(FEAT_SPEv1p2) || !IsFeatureImplemented(FEAT_PMUv3p7) then return FALSE;
    if PMBSR_EL1.S != '1' || PMBLIMITR_EL1.[E,PMFZ] != '11' then return FALSE;

    if idx == CYCLE\_COUNTER\_ID && !IsFeatureImplemented(FEAT_SPE_DPFZS) then
        // FZS does not affect the cycle counter when FEAT_SPE_DPFZS is not implemented
        return FALSE;

    if PMUCounterIsHyp(idx) then
        return MDCR_EL2.HPMFZS == '1';
    else
        return PMCR_EL0.FZS == '1';
```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEGetDataSourcePayloadSize

```
// SPEGetDataSourcePayloadSize()
// =====
// Returns the size of the Data Source payload in bytes.

integer SPEGetDataSourcePayloadSize()
    return integer IMPLEMENTATION_DEFINED "SPE Data Source packet payload size";
```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEGetEventsPayloadSize

```
// SPEGetEventsPayloadSize()
// =====
// Returns the size in bytes of the Events packet payload as an integer.

integer SPEGetEventsPayloadSize()
    integer size = integer IMPLEMENTATION_DEFINED "SPE Events packet payload size";
    return size;
```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEGetRandomBoolean

```
// SPEGetRandomBoolean()
// =====
// Returns a random or pseudo-random boolean value.

boolean SPEGetRandomBoolean();
```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEGetRandomInterval

```
// SPEGetRandomInterval()
// =====
// Returns a random or pseudo-random byte for resetting COUNT or ECOUNT.

bits(8) SPEGetRandomInterval();
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEISB

```
// SPEISB()
// =====
// Called by ISB instruction, correctly calls SPEBranch to save previous branches.

SPEISB()
    bits(64) address = PC[] + 4;
    BranchType branch_type = BranchType\_DIR;
    boolean branch_conditional = FALSE;
    boolean taken = FALSE;
    boolean is_isb = TRUE;

    SPEBranch(address, branch_type, branch_conditional, taken, is_isb);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEMaxAddr

```
constant integer SPEMaxAddr = 32;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEMaxCounters

```
constant integer SPEMaxCounters = 32;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEMaxRecordSize

```
constant integer SPEMaxRecordSize = 64;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEPostExecution

```
constant integer SPEAddrPosPCVirtual = 0;
constant integer SPEAddrPosBranchTarget = 1;
constant integer SPEAddrPosDataVirtual = 2;
constant integer SPEAddrPosDataPhysical = 3;
constant integer SPEAddrPosPrevBranchTarget = 4;
constant integer SPECounterPosTotalLatency = 0;
constant integer SPECounterPosIssueLatency = 1;
constant integer SPECounterPosTranslationLatency = 2;
boolean SPESampleInFlight = FALSE;
bits(32) SPESampleContextEL1;
boolean SPESampleContextEL1Valid;
bits(32) SPESampleContextEL2;
boolean SPESampleContextEL2Valid;
boolean SPESampleInstIsNV2 = FALSE;
bits(64) SPESamplePreviousBranchAddress;
boolean SPESamplePreviousBranchAddressValid;
bits(16) SPESampleDataSource;
boolean SPESampleDataSourceValid;
OpType SPESampleOpType;
bits(2) SPESampleClass;
bits(8) SPESampleSubclass;
boolean SPESampleSubclassValid;
bits(64) SPESampleTimestamp;
boolean SPESampleTimestampValid;
bits(64) SPESampleEvents;

// SPEPostExecution()
// =====
// Called after every executed instruction.

SPEPostExecution()
    if SPESampleInFlight then
        SPESampleInFlight = FALSE;
        PMUEvent(PMU_EVENT_SAMPLE_FEED);

        // Stop any pending counters
        for counter_index = 0 to (SPEMaxCounters - 1)
            if SPESampleCounterPending[counter_index] then
                SPEStopCounter(counter_index);

        boolean discard = FALSE;
        if HaveStatisticalProfilingvlp2() then
            discard = PMBLIMITR_EL1.FM == '10';
        if SPECollectRecord(SPESampleEvents,
                           SPESampleCounter[SPECounterPosTotalLatency],
                           SPESampleOpType) && !discard then
            SPEConstructRecord();
            if SPEBufferIsFull() then
                SPEBufferFilled();

        SPEResetSampleStorage();

// Counter storage
array [0..SPEMaxCounters-1] of integer SPESampleCounter;

array [0..SPEMaxCounters-1] of boolean SPESampleCounterValid;

array [0..SPEMaxCounters-1] of boolean SPESampleCounterPending;

// Address storage
array [0..SPEMaxAddrs-1] of bits(64) SPESampleAddress;

array [0..SPEMaxAddrs-1] of boolean SPESampleAddressValid;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEPreExecution

```
// SPEPreExecution()
// =====
// Called prior to execution, for all instructions.

SPEPreExecution()
    if StatisticalProfilingEnabled\(\) then
        PMUEvent(PMU_EVENT_SAMPLE_POP);
        if SPEToCollectSample\(\) then
            if !SPESampleInFlight then
                SPESampleInFlight = TRUE;

                // Start total latency and issue latency counters for SPE
                SPEStartCounter(SPECounterPosTotalLatency);
                SPEStartCounter(SPECounterPosIssueLatency);

                SPESampleAddContext();

                SPESampleAddAddressPCVirtual();

                // Timestamp may be collected at any point in the sampling operation.
                // Collecting prior to execution is one possible choice.
                // This choice is IMPLEMENTATION_DEFINED.
                SPESampleAddTimeStamp();
            else
                PMUEvent(PMU_EVENT_SAMPLE_COLLISION);
                PMBSR_EL1.COLL = '1';

                // Many operations are type other and not conditional, can save footprint
                // and overhead by having this as the default and not calling SPESampleAddOpOther
                // if conditional == FALSE
                SPESampleAddOpOther(FALSE);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEResetSampleCounter

```
// SPEResetSampleCounter()
// =====
// Reset PMSICR_EL1.Counter

SPEResetSampleCounter()
    PMSICR_EL1.COUNT<31:8> = PMSIRR_EL1.INTERVAL;
    if PMSIRR_EL1.RND == '1' && PMSIDR_EL1.ERnd == '0' then
        PMSICR_EL1.COUNT<7:0> = SPEGetRandomInterval();
    else
        PMSICR_EL1.COUNT<7:0> = Zeros(8);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEResetSampleStorage

```
integer SPERecordSize;

// SPEResetSampleStorage()
// =====
// Reset all variables inside sample storage.

SPEResetSampleStorage()
    // Context values
    SPESampleContextEL1 = Zeros(32);
    SPESampleContextEL1Valid = FALSE;
    SPESampleContextEL2 = Zeros(32);
    SPESampleContextEL2Valid = FALSE;

    // Counter values
    for i = 0 to (SPEMaxCounters - 1)
        SPESampleCounter[i] = 0;
        SPESampleCounterValid[i] = FALSE;
        SPESampleCounterPending[i] = FALSE;

    // Address values
    for i = 0 to (SPEMaxAddrs - 1)
        SPESampleAddressValid[i] = FALSE;
        SPESampleAddress[i] = Zeros(64);

    // Data source values
    SPESampleDataSource = Zeros(16);
    SPESampleDataSourceValid = FALSE;

    // Operation values
    SPESampleClass = Zeros(2);
    SPESampleSubclass = Zeros(8);
    SPESampleSubclassValid = FALSE;

    // Timestamp values
    SPESampleTimestamp = Zeros(64);
    SPESampleTimestampValid = FALSE;

    // Event values
    SPESampleEvents<63:48> = bits(16) IMPLEMENTATION_DEFINED "SPE EVENTS 63_48";
    SPESampleEvents<47:32> = Zeros(16);
    SPESampleEvents<31:24> = bits(8) IMPLEMENTATION_DEFINED "SPE EVENTS 31_24";
    SPESampleEvents<23:16> = Zeros(8);
    SPESampleEvents<15:12> = bits(4) IMPLEMENTATION_DEFINED "SPE EVENTS 15_12";
    SPESampleEvents<11:0> = Zeros(12);

    SPESampleInstIsNV2 = FALSE;

array [0..SPEMaxRecordSize-1] of bits(8) SPERecordData;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleAddAddressPCVirtual

```
// SPESampleAddAddressPCVirtual()
// =====
// Save the current PC address to sample storage.

SPESampleAddAddressPCVirtual()
    bits(64) this_address = ThisInstrAddr(64);
    SPESampleAddress[SPEAddrPosPCVirtual] <55:0> = this_address <55:0>;

    bit ns;
    bit nse;
    case CurrentSecurityState() of
        when SS\_Secure
            ns = '0';
            nse = '0';
        when SS\_NonSecure
            ns = '1';
            nse = '0';
        when SS\_Realm
            ns = '1';
            nse = '1';
        otherwise Unreachable();

    bits(2) el = PSTATE.EL;
    SPESampleAddress[SPEAddrPosPCVirtual] <63:56> = ns:el:nse:Zeros(4);
    SPESampleAddressValid[SPEAddrPosPCVirtual] = TRUE;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleAddContext

```
// SPESampleAddContext()
// =====
// Save contexts to sample storage if appropriate.

SPESampleAddContext()
    if CollectContextIDR1() then
        SPESampleContextEL1 = CONTEXTIDR_EL1 <31:0>;
        SPESampleContextEL1Valid = TRUE;
    if CollectContextIDR2() then
        SPESampleContextEL2 = CONTEXTIDR_EL2 <31:0>;
        SPESampleContextEL2Valid = TRUE;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleAddOpOther

```
// SPESampleAddOpOther()
// =====
// Add other operation to sample storage.

SPESampleAddOpOther(boolean conditional, boolean taken)
    SPESampleEvents <6> = if conditional && !taken then '1' else '0';
    SPESampleAddOpOther(conditional);

SPESampleAddOpOther(boolean conditional)
    SPESampleClass = '00';
    SPESampleSubclass <0> = if conditional then '1' else '0';
    SPESampleOpType = OpType\_Other;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleAddOpSVELoadStore

```
// SPESampleAddOpSVELoadStore()
// =====
// Sets the subclass of the operation type packet to Load/Store for SVE operations.

SPESampleAddOpSVELoadStore(boolean is_gather_scatter, bits(3) evl, boolean predicated,
                           boolean is_load)
    bit sg = if is_gather_scatter then '1' else '0';
    bit pred = if predicated then '1' else '0';
    bit ldst = if is_load then '0' else '1';
    SPESampleClass = '01';
    SPESampleSubclass<7:0> = sg:evl:'1':pred:'0':ldst;
    SPESampleSubclassValid = TRUE;
    SPESampleOpType = if is_load then OpType\_Load else OpType\_Store;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleAddOpSVEOther

```
// SPESampleAddOpSVEOther()
// =====
// Sets the subclass of the operation type packet to Other for SVE operations.

SPESampleAddOpSVEOther(bits(3) evl, boolean predicated, boolean floating_point)
    bit pred = if predicated then '1' else '0';
    bit fp = if floating_point then '1' else '0';
    SPESampleClass = '00';
    SPESampleSubclass<7:0> = '0':evl:'1':pred:fp:'0';
    SPESampleSubclassValid = TRUE;
    SPESampleOpType = OpType\_Other;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleAddTimeStamp

```
// SPESampleAddTimeStamp()
// =====
// Save the appropriate type of timestamp to sample storage.

SPESampleAddTimeStamp()
    TimeStamp timestamp = CollectTimeStamp();
    case timestamp of
        when TimeStamp\_None
            SPESampleTimestampValid = FALSE;
        otherwise
            SPESampleTimestampValid = TRUE;
            SPESampleTimestamp = GetTimeStamp(timestamp);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleExtendedLoadStore

```
// SPESampleExtendedLoadStore()
// =====
// Sets the subclass of the operation type packet for
// extended load/store operations.

SPESampleExtendedLoadStore(bit ar, bit excl, bit at, boolean is_load)
    SPESampleClass = '01';
    bit ldst = if is_load then '0' else '1';
    SPESampleSubclass = '000':ar:excl:at:'1':ldst;

    SPESampleSubclassValid = TRUE;

    if is_load then
        if at == '1' then
            SPESampleOpType = OpType\_LoadAtomic;
        else
            SPESampleOpType = OpType\_Load;
    else
        SPESampleOpType = OpType\_Store;
```



## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleGeneralPurposeLoadStore

```
// SPESampleGeneralPurposeLoadStore()
// =====
// Sets the subclass of the operation type packet for general
// purpose load/store operations.

SPESampleGeneralPurposeLoadStore()
    SPESampleClass = '01';

    SPESampleSubclass<7:1> = Zeros(7);
    SPESampleSubclassValid = TRUE;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleLoadStore

```
// SPESampleLoadStore()
// =====
// Called if a sample is in flight when writing or reading memory,
// indicating that the operation being sampled is in the Load, Store or atomic category.

SPESampleLoadStore(boolean is_load, AccessDescriptor accdesc, AddressDescriptor addrdesc)
    // Check if this access type should be sampled.
    if accdesc.acctype IN {AccessType\_SPE,
                          AccessType\_IFETCH,
                          AccessType\_DC,
                          AccessType\_TTW,
                          AccessType\_AT} then

        return;

    // MOPS instructions indicate which operation should be sampled before the
    // operation is executed. Has the instruction indicated that the load should be sampled?
    boolean sample_loads;
    sample_loads = SPESampleSubclass<0> == '0' && SPESampleSubclassValid;

    // Has the instruction indicated that the store should be sampled?
    boolean sample_stores;
    sample_stores = SPESampleSubclass<0> == '1' && SPESampleSubclassValid;

    // No valid data has been collected, or this is operation has specifically been selected for
    // sampling.
    if (!SPESampleSubclassValid || (sample_loads && is_load) ||
        (sample_stores && !is_load)) then
        // Data access virtual address
        SPESetDataVirtualAddress(addrdesc.vaddress);

        // Data access physical address
        if CollectPhysicalAddress() then
            SPESetDataPhysicalAddress(addrdesc, accdesc);

    if !SPESampleSubclassValid then
        // Set as unspecified load/store by default, instructions will overwrite this if it does not
        // apply to them.
        SPESampleClass = '01';
        SPESampleSubclassValid = TRUE;
        SPESampleSubclass<7:1> = '0001000';
        SPESampleSubclass<0> = if is_load then '0' else '1';
        SPESampleOpType = if is_load then OpType\_Load else OpType\_Store;

    if accdesc.acctype == AccessType\_NV2 then
        // NV2 register load/store
        SPESampleSubclass<7:1> = '0011000';
        SPESampleInstIsNV2 = TRUE;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleMemCopy

```
// SPESampleMemCopy()
// =====
// Sets the subclass of the operation type packet for Memory Copy load/store
// operations.

SPESampleMemCopy()
    // MemCopy does a read and a write. If one is filtered out, the other should be recorded.
    // If neither or both are filtered out, pick one in a (pseudo)random way.

    // Are loads allowed by filter?
    boolean loads_pass_filter = PMSFCR_EL1.FT == '1' && PMSFCR_EL1.LD == '1';
    // Are stores allowed by filter?
    boolean stores_pass_filter = PMSFCR_EL1.FT == '1' && PMSFCR_EL1.ST == '1';

    boolean record_load;
    if loads_pass_filter && !stores_pass_filter then
        // Only loads pass filter
        record_load = TRUE;
    elseif !loads_pass_filter && stores_pass_filter then
        // Only stores pass filter
        record_load = FALSE;
    else
        // Pick randomly between
        record_load = SPEGetRandomBoolean\(\);

    SPESampleClass = '01';
    bit ldst = if record_load then '0' else '1';
    SPESampleSubclass<7:0> = '0010000':ldst;
    SPESampleSubclassValid = TRUE;
    SPESampleOpType = if record_load then OpType\_Load else OpType\_Store;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleMemSet

```
// SPESampleMemSet()
// =====
// Sets the subclass of the operation type packet for Memory Set load/store
// operation.

SPESampleMemSet()
    SPESampleClass = '01';
    SPESampleSubclass<7:0> = '00100101';
    SPESampleSubclassValid = TRUE;
    SPESampleOpType = OpType\_Store;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleSIMDFPLoadStore

```
// SPESampleSIMDFPLoadStore()
// =====
// Sets the subclass of the operation type packet for SIMD & FP
// load store operations.

SPESampleSIMDFPLoadStore()
    SPESampleClass = '01';

    SPESampleSubclass<7:1> = '0000010';
    SPESampleSubclassValid = TRUE;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESetDataPhysicalAddress

```
// SPESetDataPhysicalAddress()
// =====
// Called from SampleLoadStore() to save data physical packet.

SPESetDataPhysicalAddress(AddressDescriptor addrdesc, AccessDescriptor accdesc)
    bit ns;
    bit nse;
    case addrdesc.paddress.paspace of
        when PAS\_Secure
            ns = '0';
            nse = '0';
        when PAS\_NonSecure
            ns = '1';
            nse = '0';
        when PAS\_Realm
            ns = '1';
            nse = '1';
        otherwise Unreachable();

    if HaveMTE2Ext() then
        bits(4) pat;
        if accdesc.tagchecked then
            SPESampleAddress[SPEAddrPosDataPhysical]<62> = '1'; // CH
            pat = AArch64.PhysicalTag(addrdesc.vaddress);
        else
            // CH is reset to 0 on each new packet
            // If the access is Unchecked, this is an IMPLEMENTATION_DEFINED choice
            // between 0b0000 and the Physical Address Tag
            boolean zero_unchecked;
            zero_unchecked = boolean IMPLEMENTATION_DEFINED "SPE PAT for tag unchecked access zero";
            if !zero_unchecked then
                pat = AArch64.PhysicalTag(addrdesc.vaddress);
            else
                pat = Zeros(4);
            SPESampleAddress[SPEAddrPosDataPhysical]<59:56> = pat;

    bits(56) paddr = addrdesc.paddress.address;
    SPESampleAddress[SPEAddrPosDataPhysical]<56-1:0> = paddr;
    SPESampleAddress[SPEAddrPosDataPhysical]<63> = ns;
    SPESampleAddress[SPEAddrPosDataPhysical]<60> = nse;
    SPESampleAddressValid[SPEAddrPosDataPhysical] = TRUE;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESetDataVirtualAddress

```
// SPESetDataVirtualAddress()
// =====
// Called from SampleLoadStore() to save data virtual packet.
// Also used by exclusive load/stores to save virtual addresses if exclusive monitor is lost
// before a read/write is completed.

SPESetDataVirtualAddress(bits(64) vaddress)
    bit tbi;
    tbi = EffectiveTBI(vaddress, FALSE, PSTATE.EL);
    boolean non_tbi_is_zeros;
    non_tbi_is_zeros = boolean IMPLEMENTATION_DEFINED "SPE non-tbi tag is zero";
    if tbi == '1' || !non_tbi_is_zeros then
        SPESampleAddress[SPEAddrPosDataVirtual]<63:0> = vaddress<63:0>;
    else
        SPESampleAddress[SPEAddrPosDataVirtual]<63:56> = Zeros(8);
        SPESampleAddress[SPEAddrPosDataVirtual]<55:0> = vaddress<55:0>;
    SPESampleAddressValid[SPEAddrPosDataVirtual] = TRUE;
```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEStartCounter

```
// SPEStartCounter()
// =====
// Enables incrementing of the counter at the passed index when SPECycle is called.

SPEStartCounter(integer counter_index)
    assert counter_index < SPEMaxCounters;
    SPESampleCounterPending[counter_index] = TRUE;
```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEStopCounter

```
// SPEStopCounter()
// =====
// Disables incrementing of the counter at the passed index when SPECycle is called.

SPEStopCounter(integer counter_index)
    SPESampleCounterValid[counter_index] = TRUE;
    SPESampleCounterPending[counter_index] = FALSE;
```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEToCollectSample

```
// SPEToCollectSample()
// =====
// Returns TRUE if the instruction which is about to be executed should be
// sampled. Returns FALSE otherwise.

boolean SPEToCollectSample()
    if IsZero(PMSICR_EL1.COUNT) then
        SPEResetSampleCounter();
    else
        PMSICR_EL1.COUNT = PMSICR_EL1.COUNT - 1;
        if IsZero(PMSICR_EL1.COUNT) then
            if PMSIRR_EL1.RND == '1' && PMSIDR_EL1.ERnd == '1' then
                PMSICR_EL1.ECOUNT = SPEGetRandomInterval();
            else
                return TRUE;
        if UInt(PMSICR_EL1.ECOUNT) != 0 then
            PMSICR_EL1.ECOUNT = PMSICR_EL1.ECOUNT - 1;
            if IsZero(PMSICR_EL1.ECOUNT) then
                return TRUE;
    return FALSE;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEWriteToBuffer

```
// SPEWriteToBuffer()
// =====
// Write the active record to the Profiling Buffer.

SPEWriteToBuffer()
    assert ProfilingBufferEnabled();

    // Check alignment
    boolean aligned = IsZero(PMBPTR_EL1.PTR<UInt(PMBIDR_EL1.Align)-1:0>);
    boolean ttw_fault_as_external_abort;
    ttw_fault_as_external_abort = boolean IMPLEMENTATION_DEFINED "SPE TTW fault External abort";

    FaultRecord fault;
    PhysMemRetStatus memstatus;
    AddressDescriptor addrdesc;
    AccessDescriptor accdesc;

    SecurityState owning_ss;
    bits(2) owning_el;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    accdesc = CreateAccDescSPE(owning_ss, owning_el);

    bits(64) start_vaddr = PMBPTR_EL1<63:0>;
    for i = 0 to SPERecordSize - 1
        // If a previous write did not cause an issue
        if PMBSR_EL1.S == '0' then
            (memstatus, addrdesc) = DebugMemWrite(PMBPTR_EL1<63:0>, accdesc, aligned,
                SPERecordData[i]);

            fault = addrdesc.fault;

            boolean ttw_fault;
            ttw_fault = fault.statuscode IN {Fault\_SyncExternalOnWalk, Fault\_SyncParityOnWalk};

            if IsFault(fault.statuscode) && !(ttw_fault && ttw_fault_as_external_abort) then
                DebugWriteFault(PMBPTR_EL1<63:0>, fault);
            elseif IsFault(memstatus) || (ttw_fault && ttw_fault_as_external_abort) then
                DebugWriteExternalAbort(memstatus, addrdesc, start_vaddr);

            // Move pointer if no Buffer Management Event has been caused.
            if IsZero(PMBSR_EL1.S) then
                PMBPTR_EL1 = PMBPTR_EL1 + 1;

    return;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/StatisticalProfilingEnabled

```
// StatisticalProfilingEnabled()
// =====
// Return TRUE if Statistical Profiling is Enabled in the current EL, FALSE otherwise.

boolean StatisticalProfilingEnabled()
    return StatisticalProfilingEnabled(PSTATE.EL);

// StatisticalProfilingEnabled()
// =====
// Return TRUE if Statistical Profiling is Enabled in the specified EL, FALSE otherwise.

boolean StatisticalProfilingEnabled(bits(2) el)
    if !HaveStatisticalProfiling() || UsingAArch32() || !ProfilingBufferEnabled() then
        return FALSE;

    tge_set = EL2Enabled() && HCR_EL2.TGE == '1';
    (owning_ss, owning_el) = ProfilingBufferOwner();
    if (UInt(owning_el) < UInt(el) || (tge_set && owning_el == EL1) ||
        owning_ss != SecurityStateAtEL(el)) then
        return FALSE;
    bit spe_bit;
    case el of
        when EL3    Unreachable();
        when EL2    spe_bit = PMSCR_EL2.E2SPE;
        when EL1    spe_bit = PMSCR_EL1.E1SPE;
        when EL0    spe_bit = (if tge_set then PMSCR_EL2.E0HSPE else PMSCR_EL1.E0SPE);

    return spe_bit == '1';
```

## Library pseudocode for aarch64/debug/statisticalprofiling/TimeStamp

```
// TimeStamp
// =====

enumeration TimeStamp {
    TimeStamp_None,           // No timestamp
    TimeStamp_CoreSight,      // CoreSight time (IMPLEMENTATION DEFINED)
    TimeStamp_Physical,       // Physical counter value with no offset
    TimeStamp_OffsetPhysical,  // Physical counter value minus CNTPOFF_EL2
    TimeStamp_Virtual };     // Physical counter value minus CNTVOFF_EL2
```



```

// AArch64.TakeExceptionInDebugState()
// =====
// Take an exception in Debug state to an Exception level using AArch64.

AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception_in)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
    assert target_el != EL3 || EDSCR.SDD == '0';
    ExceptionRecord except = exception_in;
    boolean sync_errors;
    if HaveIESB() then
        sync_errors = SCTLR\_EL[target_el].IESB == '1';
        if HaveDoubleFaultExt() then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && target_el == EL3);
            // SCTLR_EL[].IESB and/or SCR_EL3.NMEA (if applicable) might be ignored in Debug state.
        if !ConstrainUnpredictableBool(Unpredictable\_IESBinDebug) then
            sync_errors = FALSE;
    else
        sync_errors = FALSE;

    if HaveTME() && TSTATE.depth > 0 then
        TMFailure cause;
        case except.exceptype of
            when Exception\_SoftwareBreakpoint cause = TMFailure\_DBG;
            when Exception\_Breakpoint cause = TMFailure\_DBG;
            when Exception\_Watchpoint cause = TMFailure\_DBG;
            when Exception\_SoftwareStep cause = TMFailure\_DBG;
            otherwise cause = TMFailure\_ERR;
        FailTransaction(cause, FALSE);

    boolean brbe_source_allowed = FALSE;
    bits(64) brbe_source_address = Zeros(64);
    if HaveBRBExt() then
        brbe_source_allowed = BranchRecordAllowed(PSTATE.EL);
        brbe_source_address = bits(64) UNKNOWN;

    if !IsFeatureImplemented(FEAT_ExS) || SCTLR\_EL[target_el].EIS == '1' then
        SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();
    if from_32 && HaveSME() && PSTATE.SM == '1' then
        ResetSVEState();
    else
        MaybeZeroSVEUppers(target_el);

    AArch64.ReportException(except, target_el);

    if HaveGCS() then
        PSTATE.EXLOCK = '0'; // Effective value of GCSCR_ELx.EXLOCKEN is 0 in Debug state

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR\_ELx[] = bits(64) UNKNOWN;
    ELR\_ELx[] = bits(64) UNKNOWN;

    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    PSTATE.IL = '0';
    if from_32 then // Coming from AArch32
        PSTATE.IT = '00000000';
        PSTATE.T = '0'; // PSTATE.J is RES0
    if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
        SCTLR\_ELx{}.SPAN == '0') then
        PSTATE.PAN = '1';
    if HaveUAOExt() then PSTATE.UAO = '0';
    if HaveBTIExt() then PSTATE.BTYPE = '00';
    if HaveSSBSEExt() then PSTATE.SSBS = bit UNKNOWN;

```



```

if HaveMTEExt() then PSTATE.TCO = '1';
if IsFeatureImplemented(FEAT_EBEP) then PSTATE.PM = bit UNKNOWN;
if IsFeatureImplemented(FEAT_SEBEP) then
    PSTATE.PPEND = '0';
    ShouldSetPPEND = FALSE;

DLR_EL0 = bits(64) UNKNOWN;
DSPSR_EL0 = bits(64) UNKNOWN;

EDSCR.ERR = '1';
UpdateEDSCRFields(); // Update EDSCR processor state flags.

if sync_errors then
    SynchronizeErrors();

EndOfInstruction();

```

## Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointByteMatch

```

// AArch64.WatchpointByteMatch()
// =====

boolean AArch64.WatchpointByteMatch(integer n, bits(64) vaddress)
integer top = DebugAddrTop();
bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3; // Word or doubleword
byte_select_match = (DBGWCR_EL1[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
mask = UInt(DBGWCR_EL1[n].MASK);

// If DBGWCR_EL1[n].MASK is a nonzero value and DBGWCR_EL1[n].BAS is not set to '11111111', or
// DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
// UNPREDICTABLE.
if mask > 0 && !IsOnes(DBGWCR_EL1[n].BAS) then
    byte_select_match = ConstrainUnpredictableBool(Unpredictable\_WPMASKANDBAS);
else
    LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1)); MSB = (DBGWCR_EL1[n].BAS + LSB);
    if !IsZero(MSB AND (MSB - 1)) then // Not contiguous
        byte_select_match = ConstrainUnpredictableBool(Unpredictable\_WPBASCONTIGUOUS);
        bottom = 3; // For the whole doubleword

// If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
if mask > 0 && mask <= 2 then
    Constraint c;
    (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable\_RESWPMASK);
    assert c IN {Constraint\_DISABLED, Constraint\_NONE, Constraint\_UNKNOWN};
    case c of
        when Constraint\_DISABLED return FALSE; // Disabled
        when Constraint\_NONE mask = 0; // No masking
        // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

// When FEAT_LVA3 is not implemented, if the DBGWVR_EL1[n].RESS field bits are not a
// sign extension of the MSB of DBGWVR_EL1[n].VA, it is UNPREDICTABLE whether they
// appear to be included in the match.
if (top < 55 && !IsOnes(DBGWVR_EL1[n]<63:top>) && !IsZero(DBGWVR_EL1[n]<63:top>) &&
    ConstrainUnpredictableBool(Unpredictable\_DBGxVR\_RESS)) then
    top = 63;

boolean WVR_match;
if mask > bottom then
    WVR_match = (vaddress<top:mask> == DBGWVR_EL1[n]<top:mask>);
    // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
    if WVR_match && !IsZero(DBGWVR_EL1[n]<mask-1:bottom>) then
        WVR_match = ConstrainUnpredictableBool(Unpredictable\_WPMASKEDBITS);
else
    WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;

return (WVR_match && byte_select_match);

```

## Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointMatch

```
// AArch64.WatchpointMatch()
// =====
// Watchpoint matching in an AArch64 translation regime.

boolean AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size,
                                AccessDescriptor accdesc)
    assert !ELUsingAArch32\(S1TranslationRegime\(\)\);
    assert n < NumWatchpointsImplemented\(\);

    boolean enabled          = IsWatchpointEnabled(n);
    linked = DBGWCR_EL1[n].WT == '1';
    isbreakpnt = FALSE;
    lbnx = if Havev8p9Debug() then DBGWCR_EL1[n].LBNX else '00';
    linked_n = UInt(lbnx : DBGWCR_EL1[n].LBN);
    ssce = if HaveRME() then DBGWCR_EL1[n].SSCE else '0';
    state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, ssce, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,
                                    linked, linked_n, isbreakpnt, PC[], accdesc);

    boolean ls_match;
    case DBGWCR_EL1[n].LSC<1:0> of
        when '00' ls_match = FALSE;
        when '01' ls_match = accdesc.read;
        when '10' ls_match = accdesc.write || accdesc.acctype == AccessType\_DC;
        when '11' ls_match = TRUE;

    boolean value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch64.WatchpointByteMatch(n, vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

## Library pseudocode for aarch64/debug/watchpoint/IsWatchpointEnabled

```
// IsWatchpointEnabled()
// =====
// Returns TRUE if the effective value of DBGWCR_EL1[n].E is '1', and FALSE otherwise.

boolean IsWatchpointEnabled(integer n)
    if (n > 15 &&
        ((!HaltOnBreakpointOrWatchpoint() && !SelfHostedExtendedBPWPEEnabled()) ||
         (HaltOnBreakpointOrWatchpoint() && EDSCR2.EHBWE == '0')) then
        return FALSE;
    return DBGWCR_EL1[n].E == '1';
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.Abort

```
// AArch64.Abort()
// =====
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(bits(64) vaddress, FaultRecord fault)

    if IsDebugException(fault) then
        if fault.accessdesc.acctype == AccessType\_IFETCH then
            if UsingAArch32() && fault.debugmoe == DebugException\_VectorCatch then
                AArch64.VectorCatchException(fault);
            else
                AArch64.BreakpointException(fault);
        else
            AArch64.WatchpointException(vaddress, fault);
    elseif fault.gpcf.gpf != GPCF\_None && ReportAsGPCEException(fault) then
        TakeGPCEException(vaddress, fault);
    elseif fault.accessdesc.acctype == AccessType\_IFETCH then
        AArch64.InstructionAbort(vaddress, fault);
    else
        AArch64.DataAbort(vaddress, fault);
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.AbortSyndrome

```
// AArch64.AbortSyndrome()
// =====
// Creates an exception syndrome record for Abort and Watchpoint exceptions
//
// from an AArch64 translation regime.

ExceptionRecord AArch64.AbortSyndrome(Exception exceptype, FaultRecord fault,
                                       bits(64) vaddress, bits(2) target_el)
    except = ExceptionSyndrome(exceptype);

    if (!HavePFAR() ||
        !IsExternalSyncAbort(fault) ||
        (EL2Enabled() && HCR_EL2.VM == '1' && target_el == EL1)) then
        except.pavalid = FALSE;
    else
        except.pavalid = boolean IMPLEMENTATION_DEFINED "PFAR_ELx is valid";

    (except.syndrome, except.syndrome2) = AArch64.FaultSyndrome(exceptype, fault, except.pavalid,
                                                                vaddress);

    if fault.statuscode == Fault_TagCheck then
        if HaveMTE4Ext() then
            except.vaddress = ZeroExtend(vaddress, 64);
        else
            except.vaddress = bits(4) UNKNOWN : vaddress<59:0>;
    else
        except.vaddress = ZeroExtend(vaddress, 64);

    if IPAValid(fault) then
        except.ipavalid = TRUE;
        except.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        except.ipaddress = fault.ipaddress.address;
    else
        except.ipavalid = FALSE;

    return except;
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.CheckPCAlignment

```
// AArch64.CheckPCAlignment()
// =====

AArch64.CheckPCAlignment()
    bits(64) pc = ThisInstrAddr(64);

    if pc<1:0> != '00' then
        AArch64.PCAlignmentFault();
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.DataAbort

```
// AArch64.DataAbort()
// =====

AArch64.DataAbort(bits(64) vaddress, FaultRecord fault)
    bits(2) target_el;
    if IsExternalAbort(fault) then
        target_el = AArch64.SyncExternalAbortTarget(fault);
    else
        route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
            (HCR_EL2.TGE == '1' ||
            (HaveRME() && fault.gpcf.gpf == GPCF\_Fail && HCR_EL2.GPF == '1') ||
            (HaveNV2Ext() && fault.accessdesc.acctype == AccessType\_NV2) ||
            IsSecondStage(fault)));

        if PSTATE.EL == EL3 then
            target_el = EL3;
        elseif PSTATE.EL == EL2 || route_to_el2 then
            target_el = EL2;
        else
            target_el = EL1;

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset;

    if IsExternalAbort(fault) && AArch64.RouteToSErrorOffset(target_el) then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;

    ExceptionRecord except;
    if HaveNV2Ext() && fault.accessdesc.acctype == AccessType\_NV2 then
        except = AArch64.AbortSyndrome(Exception\_NV2DataAbort, fault, vaddress, target_el);
    else
        except = AArch64.AbortSyndrome(Exception\_DataAbort, fault, vaddress, target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.EffectiveTCF

```
// AArch64.EffectiveTCF()
// =====
// Indicate if a Tag Check Fault should cause a synchronous exception,
// be asynchronously accumulated, or have no effect on the PE.

TCFType AArch64.EffectiveTCF(bits(2) el, boolean read)
    bits(2) tcf;

    Regime regime = TranslationRegime(el);

    case regime of
        when Regime\_EL3    tcf = SCTLR_EL3.TCF;
        when Regime\_EL2    tcf = SCTLR_EL2.TCF;
        when Regime\_EL20   tcf = if el == EL0 then SCTLR_EL2.TCF0 else SCTLR_EL2.TCF;
        when Regime\_EL10   tcf = if el == EL0 then SCTLR_EL1.TCF0 else SCTLR_EL1.TCF;
        otherwise          Unreachable();

    if tcf == '11' then // Reserved value
        if !HaveMTEAsymFaultExt() then
            (-,tcf) = ConstrainUnpredictableBits(Unpredictable\_RESTCF, 2);

    case tcf of
        when '00' // Tag Check Faults have no effect on the PE
            return TCFType\_Ignore;
        when '01' // Tag Check Faults cause a synchronous exception
            return TCFType\_Sync;
        when '10'
            if HaveMTEAsymExt() then
                // If asynchronous faults are implemented,
                // Tag Check Faults are asynchronously accumulated
                return TCFType\_Async;
            else
                // Otherwise, Tag Check Faults have no effect on the PE
                return TCFType\_Ignore;
        when '11'
            if HaveMTEAsymFaultExt() then
                // Tag Check Faults cause a synchronous exception on reads or on
                // a read/write access, and are asynchronously accumulated on writes
                if read then
                    return TCFType\_Sync;
                else
                    return TCFType\_Async;
            else
                // Otherwise, Tag Check Faults have no effect on the PE
                return TCFType\_Ignore;
        otherwise
            Unreachable();
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.InstructionAbort

```
// AArch64.InstructionAbort()
// =====

AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)
// External aborts on instruction fetch must be taken synchronously
if HaveDoubleFaultExt() then assert fault.statuscode != Fault\_AsyncExternal;

bits(2) target_el;
if IsExternalAbort(fault) then
    target_el = AArch64.SyncExternalAbortTarget(fault);
else
    route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
        (HCR_EL2.TGE == '1' ||
        (HaveRME() && fault.gpcf.gpf == GPCF\_Fail && HCR_EL2.GPF == '1') ||
        IsSecondStage(fault)));

    if PSTATE.EL == EL3 then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_el2 then
        target_el = EL2;
    else
        target_el = EL1;

bits(64) preferred_exception_return = ThisInstrAddr(64);
integer vect_offset;

if IsExternalAbort(fault) && AArch64.RouteToSErrorOffset(target_el) then
    vect_offset = 0x180;
else
    vect_offset = 0x0;

ExceptionRecord except = AArch64.AbortSyndrome(Exception\_InstructionAbort, fault,
    vaddress, target_el);
AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.PCAlignmentFault

```
// AArch64.PCAlignmentFault()
// =====
// Called on unaligned program counter in AArch64 state.

AArch64.PCAlignmentFault()

bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x0;

except = ExceptionSyndrome(Exception\_PCAlignment);
except.vaddress = ThisInstrAddr(64);
bits(2) target_el = EL1;
if UInt(PSTATE.EL) > UInt(EL1) then
    target_el = PSTATE.EL;
elsif EL2Enabled() && HCR_EL2.TGE == '1' then
    target_el = EL2;
AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```



```

// AArch64.PhysicalErrorTarget()
// =====
// Returns a tuple of whether SError exception can be taken and, if so, the target Exception level.

(boolean, bits(2)) AArch64.PhysicalErrorTarget()
    boolean route_to_el3;
    boolean route_to_el2;

    // The exception is explicitly routed to EL3.
    if PSTATE.EL != EL3 then
        route_to_el3 = (HaveEL(EL3) && EffectiveEA() == '1');
    else
        route_to_el3 = FALSE;

    // The exception is explicitly routed to EL2.
    if !route_to_el3 && EL2Enabled() && PSTATE.EL == EL1 then
        route_to_el2 = (HCR_EL2.AMO == '1');
    elsif !route_to_el3 && EL2Enabled() && PSTATE.EL == EL0 then
        route_to_el2 = (!IsInHost() && HCR_EL2.<TGE,AMO> != '00');
    else
        route_to_el2 = FALSE;

    // The exception is "masked".
    boolean masked;
    case PSTATE.EL of
        when EL3
            masked = (EffectiveEA() == '0' || PSTATE.A == '1');
        when EL2
            masked = (!route_to_el3 &&
                (HCR_EL2.<TGE,AMO> == '00' || PSTATE.A == '1'));
        when EL1, EL0
            masked = (!route_to_el3 && !route_to_el2 && PSTATE.A == '1');

    // When FEAT_DoubleFault or FEAT_DoubleFault2 is implemented, the mask might be overridden.
    if HaveDoubleFault2Ext() then
        bit nmea_bit;
        case PSTATE.EL of
            when EL3
                nmea_bit = SCR_EL3.NMEA;
            when EL2
                nmea_bit = if IsSCTLR2EL2Enabled() then SCTLR2_EL2.NMEA else '0';
            when EL1
                nmea_bit = if IsSCTLR2EL1Enabled() then SCTLR2_EL1.NMEA else '0';
            when EL0
                if IsInHost() then
                    nmea_bit = if IsSCTLR2EL2Enabled() then SCTLR2_EL2.NMEA else '0';
                else
                    nmea_bit = if IsSCTLR2EL1Enabled() then SCTLR2_EL1.NMEA else '0';
        masked = masked && (nmea_bit == '0');

    elsif HaveDoubleFaultExt() && PSTATE.EL == EL3 then
        bit nmea_bit = SCR_EL3.NMEA AND EffectiveEA();
        masked = masked && (nmea_bit == '0');

    boolean route_masked_to_el3;
    boolean route_masked_to_el2;

    if HaveDoubleFault2Ext() then
        // The masked exception is routed to EL2.
        route_masked_to_el2 = (EL2Enabled() && !route_to_el3 &&
            IsHCRXEL2Enabled() && HCRX_EL2.TMEA == '1' &&
            ((PSTATE.EL == EL1 && (PSTATE.A == '1' || masked)) ||
            (PSTATE.EL == EL0 && masked && !IsInHost())));

        // The masked exception is routed to EL3.
        route_masked_to_el3 = (HaveEL(EL3) && SCR_EL3.TMEA == '1' &&
            !(route_to_el2 || route_masked_to_el2) &&
            ((PSTATE.EL IN {EL2, EL1} &&
            (PSTATE.A == '1' || masked)) ||
            (PSTATE.EL == EL0 && masked)));

```



```

else
    route_masked_to_el2 = FALSE;
    route_masked_to_el3 = FALSE;

// The exception is taken at EL3.
take_in_el3 = PSTATE.EL == EL3 && !masked;

// The exception is taken at EL2 or in the Host EL0.
take_in_el2_0 = ((PSTATE.EL == EL2 || IsInHost\(\)) &&
    !(route_to_el3 || route_masked_to_el3) && !masked);

// The exception is taken at EL1 or in the non-Host EL0.
take_in_el1_0 = ((PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost\(\))) &&
    !(route_to_el2 || route_masked_to_el2) &&
    !(route_to_el3 || route_masked_to_el3) && !masked);

bits(2) target_el;
if take_in_el3 || route_to_el3 || route_masked_to_el3 then
    masked = FALSE; target_el = EL3;
elseif take_in_el2_0 || route_to_el2 || route_masked_to_el2 then
    masked = FALSE; target_el = EL2;
elseif take_in_el1_0 then
    masked = FALSE; target_el = EL1;
else
    masked = TRUE; target_el = bits(2) UNKNOWN;

return (masked, target_el);

```

#### Library pseudocode for aarch64/exceptions/aborts/AArch64.RaiseTagCheckFault

```

// AArch64.RaiseTagCheckFault()
// =====
// Raise a Tag Check Fault exception.

AArch64.RaiseTagCheckFault(bits(64) va, FaultRecord fault)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;
    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elseif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;

    except = AArch64.AbortSyndrome(Exception\_DataAbort, fault, va, target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);

```

#### Library pseudocode for aarch64/exceptions/aborts/AArch64.ReportTagCheckFault

```

// AArch64.ReportTagCheckFault()
// =====
// Records a Tag Check Fault exception into the appropriate TFSR_ELx.

AArch64.ReportTagCheckFault(bits(2) el, bit ttbr)
    case el of
        when EL3 assert ttbr == '0'; TFSR_EL3.TF0 = '1';
        when EL2 if ttbr == '0' then TFSR_EL2.TF0 = '1'; else TFSR_EL2.TF1 = '1';
        when EL1 if ttbr == '0' then TFSR_EL1.TF0 = '1'; else TFSR_EL1.TF1 = '1';
        when EL0 if ttbr == '0' then TFSR_EL0.TF0 = '1'; else TFSR_EL0.TF1 = '1';

```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.RouteToSErrorOffset

```
// AArch64.RouteToSErrorOffset()
// =====
// Returns TRUE if synchronous External abort exceptions are taken to the
// appropriate SError vector offset, and FALSE otherwise.

boolean AArch64.RouteToSErrorOffset(bits(2) target_el)
    if !HaveDoubleFaultExt() then return FALSE;

    bit ease_bit;
    case target_el of
        when EL3
            ease_bit = SCR_EL3.EASE;
        when EL2
            if HaveDoubleFault2Ext() && IsSCTLR2EL2Enabled() then
                ease_bit = SCTLR2_EL2.EASE;
            else
                ease_bit = '0';
        when EL1
            if HaveDoubleFault2Ext() && IsSCTLR2EL1Enabled() then
                ease_bit = SCTLR2_EL1.EASE;
            else
                ease_bit = '0';

    return (ease_bit == '1');
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.SPAlignmentFault

```
// AArch64.SPAlignmentFault()
// =====
// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception\_SPAlignment);

    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```



```

// AArch64.SyncExternalAbortTarget()
// =====
// Returns the target Exception level for a Synchronous External
// Data or Instruction Abort.

bits(2) AArch64.SyncExternalAbortTarget(FaultRecord fault)
    boolean route_to_el3;

    // The exception is explicitly routed to EL3
    if PSTATE.EL != EL3 then
        route_to_el3 = (HaveEL(EL3) && EffectiveEA() == '1');
    else
        route_to_el3 = FALSE;

    // The exception is explicitly routed to EL2
    bit tea_bit = (if HaveRASExt() && EL2Enabled() then HCR_EL2.TEA else '0');

    boolean route_to_el2;
    if !route_to_el3 && EL2Enabled() && PSTATE.EL == EL1 then
        route_to_el2 = (tea_bit == '1' ||
            fault.accessdesc.acctype == AccessType\_NV2 ||
            IsSecondStage(fault));

    elsif !route_to_el3 && EL2Enabled() && PSTATE.EL == EL0 then
        route_to_el2 = (!IsInHost() && (HCR_EL2.TGE == '1' || tea_bit == '1' ||
            IsSecondStage(fault)));

    else
        route_to_el2 = FALSE;

    boolean route_masked_to_el3;
    boolean route_masked_to_el2;

    if HaveDoubleFault2Ext() then
        // The masked exception is routed to EL2
        route_masked_to_el2 = (EL2Enabled() && !route_to_el3 &&
            (PSTATE.EL == EL1 && PSTATE.A == '1') &&
            IsHCRXEL2Enabled() && HCRX_EL2.TMEA == '1');

        // The masked exception is routed to EL3
        route_masked_to_el3 = (HaveEL(EL3) &&
            !(route_to_el2 || route_masked_to_el2) &&
            (PSTATE.EL IN {EL2, EL1} && PSTATE.A == '1') &&
            SCR_EL3.TMEA == '1');

    else
        route_masked_to_el2 = FALSE;
        route_masked_to_el3 = FALSE;

    // The exception is taken at EL3
    take_in_el3 = PSTATE.EL == EL3;

    // The exception is taken at EL2 or in the Host EL0
    take_in_el2_0 = ((PSTATE.EL == EL2 || IsInHost()) &&
        !(route_to_el3 || route_masked_to_el3));

    // The exception is taken at EL1 or in the non-Host EL0
    take_in_el1_0 = ((PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost())) &&
        !(route_to_el2 || route_masked_to_el2) &&
        !(route_to_el3 || route_masked_to_el3));

    bits(2) target_el;
    if take_in_el3 || route_to_el3 || route_masked_to_el3 then
        target_el = EL3;
    elsif take_in_el2_0 || route_to_el2 || route_masked_to_el2 then
        target_el = EL2;
    elsif take_in_el1_0 then
        target_el = EL1;
    else
        assert(FALSE);

    return target_el;

```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.TagCheckFault

```
// AArch64.TagCheckFault()
// =====
// Handle a Tag Check Fault condition.

AArch64.TagCheckFault(bits(64) vaddress, AccessDescriptor accdesc)
    TCFType tcftype = AArch64.EffectiveTCF(accdesc.el, accdesc.read);

    case tcftype of
        when TCFType\_Sync
            FaultRecord fault = NoFault();
            fault.accessdesc = accdesc;
            fault.write      = accdesc.write;
            fault.statuscode = Fault\_TagCheck;
            AArch64.RaiseTagCheckFault(vaddress, fault);
        when TCFType\_Async
            AArch64.ReportTagCheckFault(accdesc.el, vaddress<55>);
        when TCFType\_Ignore
            return;
        otherwise
            Unreachable();
```

## Library pseudocode for aarch64/exceptions/aborts/BranchTargetException

```
// BranchTargetException()
// =====
// Raise branch target exception.

AArch64.BranchTargetException(bits(52) vaddress)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception\_BranchTarget);
    except.syndrome<1:0> = PSTATE.BTYPE;
    except.syndrome<24:2> = Zeros(23); // RES0

    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/aborts/TCFType

```
// TCFType
// =====

enumeration TCFType { TCFType\_Sync, TCFType\_Async, TCFType\_Ignore };
```



```

// TakeGPCEException()
// =====
// Report Granule Protection Exception faults

TakeGPCEException(bits(64) vaddress, FaultRecord fault)
    assert HaveRME();
    assert HaveAtomicExt();
    assert HaveAccessFlagUpdateExt();
    assert HaveDirtyBitModifierExt();
    assert HaveDoubleFaultExt();

    ExceptionRecord except;

    except.exceptype = Exception\_GPC;
    except.vaddress = ZeroExtend(vaddress, 64);
    except.paddress = fault.paddress;
    except.pavalid = TRUE;

    if IPAValid(fault) then
        except.ipavalid = TRUE;
        except.NS = if fault.ipaddress.paspace == PAS\_NonSecure then '1' else '0';
        except.ipaddress = fault.ipaddress.address;
    else
        except.ipavalid = FALSE;

    if fault.accessdesc.acctype == AccessType\_GCS then
        except.syndrome<8> = '1'; //GCS

    // Populate the fields grouped in ISS
    except.syndrome<24:22> = Zeros(3); // RES0
    except.syndrome<21> = if fault.gpcfs2walk then '1' else '0'; // S2PTW
    if fault.accessdesc.acctype == AccessType\_IFETCH then
        except.syndrome<20> = '1'; // InD
    else
        except.syndrome<20> = '0'; // InD
    except.syndrome<19:14> = EncodeGPCSC(fault.gpcf); // GPCSC
    if HaveNV2Ext() && fault.accessdesc.acctype == AccessType\_NV2 then
        except.syndrome<13> = '1'; // VNCR
    else
        except.syndrome<13> = '0'; // VNCR
    except.syndrome<12:11> = '00'; // RES0
    except.syndrome<10:9> = '00'; // RES0

    if fault.accessdesc.acctype IN {AccessType\_DC, AccessType\_IC, AccessType\_AT} then
        except.syndrome<8> = '1'; // CM
    else
        except.syndrome<8> = '0'; // CM

    except.syndrome<7> = if fault.s2fslwalk then '1' else '0'; // S1PTW

    if fault.accessdesc.acctype IN {AccessType\_DC, AccessType\_IC, AccessType\_AT} then
        except.syndrome<6> = '1'; // WnR
    elseif fault.statuscode IN {Fault\_HWUpdateAccessFlag, Fault\_Exclusive} then
        except.syndrome<6> = bit UNKNOWN; // WnR
    elseif fault.accessdesc.atomicop && IsExternalAbort(fault) then
        except.syndrome<6> = bit UNKNOWN; // WnR
    else
        except.syndrome<6> = if fault.write then '1' else '0'; // WnR

    except.syndrome<5:0> = EncodeLDFSC(fault.statuscode, fault.level); // xFSC

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    bits(2) target_el = EL3;

    integer vect_offset;
    if IsExternalAbort(fault) && AArch64.RouteToSErrorOffset(target_el) then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;

```

```
AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

### Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalFIQException

```
// AArch64.TakePhysicalFIQException()
// =====

AArch64.TakePhysicalFIQException()

    route_to_el3 = HaveEL\(EL3\) && SCR_EL3.FIQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled\(\) &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x100;
    except = ExceptionSyndrome(Exception\_FIQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

### Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalIRQException

```
// AArch64.TakePhysicalIRQException()
// =====
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()

    route_to_el3 = HaveEL\(EL3\) && SCR_EL3.IRQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled\(\) &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x80;

    except = ExceptionSyndrome(Exception\_IRQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```



## Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalSErrorException

```
// AArch64.TakePhysicalSErrorException()
// =====

AArch64.TakePhysicalSErrorException(boolean implicit_esb)

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1')));
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x180;

    bits(2) target_el;
    if PSTATE.EL == EL3 || route_to_el3 then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_el2 then
        target_el = EL2;
    else
        target_el = EL1;

    except = ExceptionSyndrome(Exception_SError);
    bits(25) syndrome = AArch64.PhysicalErrorSyndrome(implicit_esb);
    if IsSErrorEdgeTriggered() then
        ClearPendingPhysicalSError();
    except.syndrome = syndrome;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualFIQException

```
// AArch64.TakeVirtualFIQException()
// =====

AArch64.TakeVirtualFIQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1'; // Virtual IRQ enabled if TGE==0 and FMO==1

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x100;

    except = ExceptionSyndrome(Exception_FIQ);

    AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualIRQException

```
// AArch64.TakeVirtualIRQException()
// =====

AArch64.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1'; // Virtual IRQ enabled if TGE==0 and IMO==1

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x80;

    except = ExceptionSyndrome(Exception_IRQ);

    AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualSErrorException

```
// AArch64.TakeVirtualSErrorException()
// =====

AArch64.TakeVirtualSErrorException()

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1'; // Virtual SError enabled if TGE==0 and AMO==1

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x180;
    except = ExceptionSyndrome(Exception_SError);

    if HaveRASExt() then
        except.syndrome<24> = VSESR_EL2.IDS;
        except.syndrome<23:0> = VSESR_EL2.ISS;
    else
        bits(25) syndrome = bits(25) IMPLEMENTATION_DEFINED "Virtual SError syndrome";
        impdef_syndrome = syndrome<24> == '1';
        if impdef_syndrome then except.syndrome = syndrome;

    ClearPendingVirtualSError();
    AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/debug/AArch64.BreakpointException

```
// AArch64.BreakpointException()
// =====

AArch64.BreakpointException(FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    bits(2) target_el;
    vect_offset = 0x0;
    target_el = if (PSTATE.EL == EL2 || route_to_el2) then EL2 else EL1;

    vaddress = bits(64) UNKNOWN;
    except = AArch64.AbortSyndrome(Exception_Breakpoint, fault, vaddress, target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/debug/AArch64.SoftwareBreakpoint

```
// AArch64.SoftwareBreakpoint()
// =====

AArch64.SoftwareBreakpoint(bits(16) immediate)

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} &&
        EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_SoftwareBreakpoint);
    except.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/debug/AArch64.SoftwareStepException

```
// AArch64.SoftwareStepException()
// =====

AArch64.SoftwareStepException()
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception\_SoftwareStep);
    if SoftwareStep\_DidNotStep() then
        except.syndrome<24> = '0';
    else
        except.syndrome<24> = '1';
        except.syndrome<6> = if SoftwareStep\_SteppedEX() then '1' else '0';
    except.syndrome<5:0> = '100010'; // IFSC = Debug Exception

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/debug/AArch64.VectorCatchException

```
// AArch64.VectorCatchException()
// =====
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.

AArch64.VectorCatchException(FaultRecord fault)
    assert PSTATE.EL != EL2;
    assert EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    except = AArch64.AbortSyndrome(Exception\_VectorCatch, fault, vaddress, EL2);

    AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/debug/AArch64.WatchpointException

```
// AArch64.WatchpointException()
// =====

AArch64.WatchpointException(bits(64) vaddress, FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    bits(2) target_el;
    vect_offset = 0x0;
    target_el = if (PSTATE.EL == EL2 || route_to_el2) then EL2 else EL1;

    ExceptionRecord except;
    if HaveNV2Ext() && fault.accessdesc.acctype == AccessType\_NV2 then
        except = AArch64.AbortSyndrome(Exception\_NV2Watchpoint, fault, vaddress, target_el);
    else
        except = AArch64.AbortSyndrome(Exception\_Watchpoint, fault, vaddress, target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/exceptions/AArch64.ExceptionClass

```
// AArch64.ExceptionClass()
// =====
// Returns the Exception Class and Instruction Length fields to be reported in ESR

(integer,bit) AArch64.ExceptionClass(Exception exceptype, bits(2) target_el)

    il_is_valid = TRUE;
    from_32 = UsingAArch32();
    integer ec;
    case exceptype of
        when Exception Uncategorized          ec = 0x00; il_is_valid = FALSE;
        when Exception WfxTrap                  ec = 0x01;
        when Exception CP15RRTTrap              ec = 0x03; assert from_32;
        when Exception CP15RRTTrap              ec = 0x04; assert from_32;
        when Exception CP14RTTrap               ec = 0x05; assert from_32;
        when Exception CP14DTTrap               ec = 0x06; assert from_32;
        when Exception AdvSIMDFPAccessTrap      ec = 0x07;
        when Exception FPIDTrap                  ec = 0x08;
        when Exception PACTrap                   ec = 0x09;
        when Exception LDST64BTrap               ec = 0x0A;
        when Exception TSTARTAccessTrap         ec = 0x1B;
        when Exception GPC                       ec = 0x1E;
        when Exception CP14RRTTrap               ec = 0x0C; assert from_32;
        when Exception BranchTarget             ec = 0x0D;
        when Exception IllegalState             ec = 0x0E; il_is_valid = FALSE;
        when Exception SupervisorCall           ec = 0x11;
        when Exception HypervisorCall           ec = 0x12;
        when Exception MonitorCall             ec = 0x13;
        when Exception SystemRegisterTrap       ec = 0x18; assert !from_32;
        when Exception SystemRegister128Trap    ec = 0x14; assert !from_32;
        when Exception SVEAccessTrap            ec = 0x19; assert !from_32;
        when Exception ERetTrap                 ec = 0x1A; assert !from_32;
        when Exception PACFail                  ec = 0x1C; assert !from_32;
        when Exception SMEAccessTrap            ec = 0x1D; assert !from_32;
        when Exception InstructionAbort         ec = 0x20; il_is_valid = FALSE;
        when Exception PCAlignment              ec = 0x22; il_is_valid = FALSE;
        when Exception DataAbort                ec = 0x24;
        when Exception NV2DataAbort             ec = 0x25;
        when Exception SPAlignment              ec = 0x26; il_is_valid = FALSE; assert !from_32;
        when Exception MemCpyMemSet              ec = 0x27;
        when Exception GCSFail                   ec = 0x2D; assert !from_32;
        when Exception FPTrappedException       ec = 0x28;
        when Exception SError                   ec = 0x2F; il_is_valid = FALSE;
        when Exception Breakpoint               ec = 0x30; il_is_valid = FALSE;
        when Exception SoftwareStep             ec = 0x32; il_is_valid = FALSE;
        when Exception Watchpoint               ec = 0x34; il_is_valid = FALSE;
        when Exception NV2Watchpoint            ec = 0x35; il_is_valid = FALSE;
        when Exception SoftwareBreakpoint       ec = 0x38;
        when Exception VectorCatch              ec = 0x3A; il_is_valid = FALSE; assert from_32;
        when Exception PMU                      ec = 0x3D;
        otherwise                               Unreachable();

    if ec IN {0x20,0x24,0x30,0x32,0x34} && target_el == PSTATE.EL then
        ec = ec + 1;

    if ec IN {0x11,0x12,0x13,0x28,0x38} && !from_32 then
        ec = ec + 4;

    bit il;
    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';
    assert from_32 || il == '1'; // AArch64 instructions always 32-bit

    return (ec,il);
```

## Library pseudocode for aarch64/exceptions/exceptions/AArch64.ReportException

```
// AArch64.ReportException()
// =====
// Report syndrome information for exception taken to AArch64 state.

AArch64.ReportException(ExceptionRecord except, bits(2) target_el)

    Exception exceptype = except.exceptype;

    (ec,il) = AArch64.ExceptionClass(exceptype, target_el);
    iss = except.syndrome;
    iss2 = except.syndrome2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    ESR_EL[target_el] = (Zeros(8) : // <63:56>
                        iss2      : // <55:32>
                        ec<5:0>   : // <31:26>
                        il        : // <25>
                        iss);      // <24:0>

    if exceptype IN {
        Exception_InstructionAbort,
        Exception_PCAlignment,
        Exception_DataAbort,
        Exception_NV2DataAbort,
        Exception_NV2Watchpoint,
        Exception_GPC,
        Exception_Watchpoint
    } then
        FAR_EL[target_el] = except.vaddress;
    else
        FAR_EL[target_el] = bits(64) UNKNOWN;

    if except.ipavalid then
        HPFAR_EL2<47:4> = except.ipaddress<55:12>;
        if IsSecureEL2Enabled() && CurrentSecurityState() == SS_Secure then
            HPFAR_EL2.NS = except.NS;
        else
            HPFAR_EL2.NS = '0';
    elsif target_el == EL2 then
        HPFAR_EL2<47:4> = bits(44) UNKNOWN;

    if except.pavalid then
        bits(64) faultaddr = ZeroExtend(except.paddress.address, 64);
        if HaveRME() then
            case except.paddress.paspace of
                when PAS_Secure      faultaddr<63:62> = '00';
                when PAS_NonSecure   faultaddr<63:62> = '10';
                when PAS_Root        faultaddr<63:62> = '01';
                when PAS_Realm       faultaddr<63:62> = '11';
            if exceptype == Exception_GPC then
                faultaddr<11:0> = Zeros(12);
        else
            faultaddr<63> = if except.paddress.paspace == PAS_NonSecure then '1' else '0';
            PFAR_EL[target_el] = faultaddr;
    elsif HavePFAR() || (HaveRME() && target_el == EL3) then
        PFAR_EL[target_el] = bits(64) UNKNOWN;
    return;
```

## Library pseudocode for aarch64/exceptions/exceptions/AArch64.ResetControlRegisters

```
// AArch64.ResetControlRegisters()
// =====
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.

AArch64.ResetControlRegisters(boolean cold_reset);
```

## Library pseudocode for aarch64/exceptions/exceptions/AArch64.TakeReset

```
// AArch64.TakeReset()
// =====
// Reset into AArch64 state

AArch64.TakeReset(boolean cold_reset)
    assert HaveAArch64();

    // Enter the highest implemented Exception level in AArch64 state
    PSTATE.nRW = '0';
    if HaveEL(EL3) then
        PSTATE.EL = EL3;
    elsif HaveEL(EL2) then
        PSTATE.EL = EL2;
    else
        PSTATE.EL = EL1;

    // Reset System registers
    // and other system components
    AArch64.ResetControlRegisters(cold_reset);

    // Reset all other PSTATE fields
    PSTATE.SP = '1'; // Select stack pointer
    PSTATE.<D,A,I,F> = '1111'; // All asynchronous exceptions masked
    PSTATE.SS = '0'; // Clear software step bit
    PSTATE.DIT = '0'; // PSTATE.DIT is reset to 0 when resetting into AArch64
    PSTATE.IL = '0'; // Clear Illegal Execution state bit

    if HaveTME() then TSTATE.depth = 0; // Non-transactional state

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64.ResetGeneralRegisters();
    AArch64.ResetSIMDFPRegisters();
    AArch64.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(64) rv; // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        rv = RVBAR_EL3;
    elsif HaveEL(EL2) then
        rv = RVBAR_EL2;
    else
        rv = RVBAR_EL1;

    // The reset vector must be correctly aligned
    assert IsZero(rv<63:AArch64.PAMax()>) && IsZero(rv<1:0>);

    boolean branch_conditional = FALSE;
    EDPRSR.R = '0'; // Leaving Reset State.
    BranchTo(rv, BranchType\_RESET, branch_conditional);
```

## Library pseudocode for aarch64/exceptions/ieeeefp/AArch64.FPTrappedException

```
// AArch64.FPTrappedException()
// =====

AArch64.FPTrappedException(boolean is_ase, bits(8) accumulated_exceptions)
    except = ExceptionSyndrome(Exception\_FPTrappedException);
    if is_ase then
        if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
            except.syndrome<23> = '1'; // TFV
        else
            except.syndrome<23> = '0'; // TFV
    else
        except.syndrome<23> = '1'; // TFV
    except.syndrome<10:8> = bits(3) UNKNOWN; // VECITR
    if except.syndrome<23> == '1' then
        except.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
    else
        except.syndrome<7,4:0> = bits(6) UNKNOWN;

    route_to_el2 = EL2Enabled() && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallHypervisor

```
// AArch64.CallHypervisor()
// =====
// Performs a HVC call

AArch64.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception\_HypervisorCall);
    except.syndrome<15:0> = immediate;

    if PSTATE.EL == EL3 then
        AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSecureMonitor

```
// AArch64.CallSecureMonitor()
// =====

AArch64.CallSecureMonitor(bits(16) immediate)
    assert HaveEL\(EL3\) && !ELUsingAArch32\(EL3\);
    if UsingAArch32\(\) then AArch32.ITAdvance\(\);
    HSAdvance\(\);
    SSAdvance\(\);
    bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome\(Exception\_MonitorCall\);
    except.syndrome<15:0> = immediate;

    AArch64.TakeException\(EL3, except, preferred\_exception\_return, vect\_offset\);
```

## Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSupervisor

```
// AArch64.CallSupervisor()
// =====
// Calls the Supervisor

AArch64.CallSupervisor(bits(16) immediate)
    if UsingAArch32\(\) then AArch32.ITAdvance\(\);
    SSAdvance\(\);
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled\(\) && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome\(Exception\_SupervisorCall\);
    except.syndrome<15:0> = immediate;

    if UInt\(PSTATE.EL\) > UInt\(EL1\) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException\(EL2, except, preferred\_exception\_return, vect\_offset\);
    else
        AArch64.TakeException\(EL1, except, preferred\_exception\_return, vect\_offset\);
```





```

// AArch64.TakeException()
// =====
// Take an exception to an Exception level using AArch64.

AArch64.TakeException(bits(2) target_el, ExceptionRecord exception_in,
                      bits(64) preferred_exception_return, integer vect_offset_in)
assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
if Halted() then
    AArch64.TakeExceptionInDebugState(target_el, exception_in);
    return;
ExceptionRecord except = exception_in;
boolean sync_errors;
boolean iesb_req;
if HaveIESB() then
    sync_errors = SCTLR\_EL[target_el].IESB == '1';
    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && target_el == EL3);
    if sync_errors && InsertIESBBeforeException(target_el) then
        SynchronizeErrors();
        iesb_req = FALSE;
        sync_errors = FALSE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
else
    sync_errors = FALSE;

if HaveTME() && TSTATE.depth > 0 then
    TMFailure cause;
    case except.exception_type of
        when Exception\_SoftwareBreakpoint cause = TMFailure\_DBG;
        when Exception\_Breakpoint cause = TMFailure\_DBG;
        when Exception\_Watchpoint cause = TMFailure\_DBG;
        when Exception\_SoftwareStep cause = TMFailure\_DBG;
        otherwise cause = TMFailure\_ERR;
    FailTransaction(cause, FALSE);

boolean brbe_source_allowed = FALSE;
bits(64) brbe_source_address = Zeros(64);
if HaveBRBExt() then
    brbe_source_allowed = BranchRecordAllowed(PSTATE.EL);
    brbe_source_address = preferred_exception_return;

if !IsFeatureImplemented(FEAT_ExS) || SCTLR\_EL[target_el].EIS == '1' then
    SynchronizeContext();

// If coming from AArch32 state, the top parts of the X[] registers might be set to zero
from_32 = UsingAArch32();
if from_32 then AArch64.MaybeZeroRegisterUppers();
if from_32 && HaveSME() && PSTATE.SM == '1' then
    ResetSVEState();
else
    MaybeZeroSVEUppers(target_el);

integer vect_offset = vect_offset_in;
if UInt(target_el) > UInt(PSTATE.EL) then
    boolean lower_32;
    if target_el == EL3 then
        if EL2Enabled() then
            lower_32 = ELUsingAArch32(EL2);
        else
            lower_32 = ELUsingAArch32(EL1);
    elsif IsInHost() && PSTATE.EL == EL0 && target_el == EL2 then
        lower_32 = ELUsingAArch32(EL0);
    else
        lower_32 = ELUsingAArch32(target_el - 1);
    vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);

elsif PSTATE.SP == '1' then
    vect_offset = vect_offset + 0x200;

bits(64) spsr = GetPSRFromPSTATE(AArch64\_NonDebugState, 64);

```

```

if PSTATE.EL == EL1 && target_el == EL1 && EL2Enabled() then
    if HaveNV2Ext() && (HCR_EL2.<NV,NV1,NV2> == '100' || HCR_EL2.<NV,NV1,NV2> == '111') then
        spsr<3:2> = '10';
    else
        if HaveNVExt() && HCR_EL2.<NV,NV1> == '10' then
            spsr<3:2> = '10';

if HaveBTIExt() && !UsingAArch32() then
    boolean zero_btype;
    // SPSR_ELx[].BTYPE is only guaranteed valid for these exception types
    if except.exception IN {Exception\_SError, Exception\_IRQ, Exception\_FIQ,
Exception\_SoftwareStep, Exception\_PCAalignment,
Exception\_InstructionAbort, Exception\_Breakpoint,
Exception\_VectorCatch, Exception\_SoftwareBreakpoint,
Exception\_IllegalState, Exception\_BranchTarget} then
        zero_btype = FALSE;
    else
        zero_btype = ConstrainUnpredictableBool(Unpredictable\_ZEROBTYPE);
    if zero_btype then spsr<11:10> = '00';

if HaveNV2Ext() && except.exception == Exception\_NV2DataAbort && target_el == EL3 then
    // External aborts are configured to be taken to EL3
    except.exception = Exception\_DataAbort;
if !(except.exception IN {Exception\_IRQ, Exception\_FIQ}) then
    AArch64.ReportException(except, target_el);

if HaveBRBExt() then
    bits(64) brbe_target_address = VBAR\_EL[target_el]<63:11>:vect_offset<10:0>;
    BRBEException(except, brbe_source_allowed, brbe_source_address,
        brbe_target_address, target_el,
        except.trappedsyscallinst);

if HaveGCS() then
    if PSTATE.EL == target_el then
        if GetCurrentEXLOCKEN() then
            PSTATE.EXLOCK = '1';
        else
            PSTATE.EXLOCK = '0';
    else
        PSTATE.EXLOCK = '0';

PSTATE.EL = target_el;
PSTATE.nRW = '0';
PSTATE.SP = '1';

SPSR\_ELx[] = spsr;
ELR\_ELx[] = preferred_exception_return;

PSTATE.SS = '0';
if HaveFeatNMI() && !ELUsingAArch32(target_el) then PSTATE.ALLINT = NOT SCTLR\_ELx[][.SPINTMASK];
PSTATE.<D,A,I,F> = '1111';
PSTATE.IL = '0';
if from_32 then // Coming from AArch32
    PSTATE.IT = '00000000';
    PSTATE.T = '0'; // PSTATE.J is RES0
if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
    SCTLR\_ELx[][.SPAN] == '0') then
    PSTATE.PAN = '1';
if HaveUAOExt() then PSTATE.UAO = '0';
if HaveBTIExt() then PSTATE.BTYPE = '00';
if HaveSSBSExt() then PSTATE.SSBS = SCTLR\_ELx[][.DSSBS];
if HaveMTEEExt() then PSTATE.TCO = '1';
if IsFeatureImplemented(FEAT_EBEP) then PSTATE.PM = '1';
if IsFeatureImplemented(FEAT_SEBEP) then
    PSTATE.PPEND = '0';
    ShouldSetPPEND = FALSE;

boolean branch_conditional = FALSE;
BranchTo(VBAR\_ELx[][<63:11>:vect_offset<10:0>, BranchType\_EXCEPTION, branch_conditional);

```

```

CheckExceptionCatch(TRUE); // Check for debug event on exception entry

if sync_errors then
    SynchronizeErrors();
    iesb_req = TRUE;
    TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

EndOfInstruction();

```

### Library pseudocode for aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrap

```

// AArch64.AArch32SystemAccessTrap()
// =====
// Trapped AARCH32 System register access.

AArch64.AArch32SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = AArch64.AArch32SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);

```



```

// AArch64.AArch32SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS,
// VMSR instructions, other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord except;

    case ec of
        when 0x0    except = ExceptionSyndrome(Exception\_Uncategorized);
        when 0x3    except = ExceptionSyndrome(Exception\_CP15RTTrap);
        when 0x4    except = ExceptionSyndrome(Exception\_CP15RRTTrap);
        when 0x5    except = ExceptionSyndrome(Exception\_CP14RTTrap);
        when 0x6    except = ExceptionSyndrome(Exception\_CP14DTTrap);
        when 0x7    except = ExceptionSyndrome(Exception\_AdvSIMDFPAccessTrap);
        when 0x8    except = ExceptionSyndrome(Exception\_FPIDTrap);
        when 0xC    except = ExceptionSyndrome(Exception\_CP14RRTTrap);
        otherwise   Unreachable();

    bits(20) iss = Zeros(20);

    if except.exceptype == Exception\_Uncategorized then
        return except;
    elseif except.exceptype IN {Exception\_FPIDTrap, Exception\_CP14RTTrap,
                               Exception\_CP15RTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        if except.exceptype != Exception\_FPIDTrap then // When trap is not for VMRS
            iss<19:17> = instr<7:5>; // opc2
            iss<16:14> = instr<23:21>; // opc1
            iss<13:10> = instr<19:16>; // CRn
            iss<4:1>   = instr<3:0>; // CRm
        else
            iss<19:17> = '000';
            iss<16:14> = '111';
            iss<13:10> = instr<19:16>; // reg
            iss<4:1>   = '0000';

        if instr<20> == '1' && instr<15:12> == '1111' then // MRC, Rt==15
            iss<9:5> = '11111';
        elseif instr<20> == '0' && instr<15:12> == '1111' then // MCR, Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
    elseif except.exceptype IN {Exception\_CP14RRTTrap, Exception\_AdvSIMDFPAccessTrap,
                               Exception\_CP15RRTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss<19:16> = instr<7:4>; // opc1
        if instr<19:16> == '1111' then // Rt2==15
            iss<14:10> = bits(5) UNKNOWN;
        else
            iss<14:10> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>;

        if instr<15:12> == '1111' then // Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
        iss<4:1>   = instr<3:0>; // CRm
    elseif except.exceptype == Exception\_CP14DTTrap then
        // Trapped LDC/STC
        iss<19:12> = instr<7:0>; // imm8
        iss<4>     = instr<23>; // U
        iss<2:1>   = instr<24,21>; // P,W
        if instr<19:16> == '1111' then // Rn==15, LDC(Literal addressing)/STC
            iss<9:5> = bits(5) UNKNOWN;
            iss<3>   = '1';
        iss<0> = instr<20>; // Direction

    except.syndrome<24:20> = ConditionSyndrome();
    except.syndrome<19:0> = iss;

```

```
return except;
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap

```
// AArch64.AdvSIMDFPAccessTrap()
// =====
// Trapped access to Advanced SIMD or FP registers due to CPACR[].

AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    ExceptionRecord except;
    vect_offset = 0x0;

    route_to_el2 = (target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1');

    if route_to_el2 then
        except = ExceptionSyndrome(Exception Uncategorized);
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        except = ExceptionSyndrome(Exception AdvSIMDFPAccessTrap);
        except.syndrome<24:20> = ConditionSyndrome();
        AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);

    return;
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckCP15InstrCoarseTraps

```
// AArch64.CheckCP15InstrCoarseTraps()
// =====
// Check for coarse-grained AArch32 traps to System registers in the
// coproc=0b1111 encoding space by HSTR_EL2, HCR_EL2, and SCTL_ELx.

AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)
    trapped_encoding = ((CRn == 9 && CRm IN {0,1,2, 5,6,7,8 }) ||
                        (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
                        (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for MRC and MCR disabled by SCTL_EL1.TIDCP.
    if (HaveFeatTIDCP1() && PSTATE.EL == EL0 && IsInHost() &&
        ELUsingAArch32(EL1) && SCTL_EL1.TIDCP == '1' && trapped_encoding) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x3);

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        // Check for MRC and MCR disabled by SCTL_EL2.TIDCP.
        if (HaveFeatTIDCP1() && PSTATE.EL == EL0 && IsInHost() &&
            SCTL_EL2.TIDCP == '1' && trapped_encoding) then
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);

    major = if nreg == 1 then CRn else CRm;
    // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR_EL2<CRn/CRm>
    // and MRC and MCR disabled by HCR_EL2.TIDCP.
    if ((!IsInHost()) && !(major IN {4,14}) && HSTR_EL2<major> == '1') ||
        (HCR_EL2.TIDCP == '1' && nreg == 1 && trapped_encoding) then
        if (PSTATE.EL == EL0 &&
            boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0") then
            UNDEFINED;
        AArch64.AArch32SystemAccessTrap(EL2, 0x3);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled

```
// AArch64.CheckFPAdvSIMDEnabled()
// =====

AArch64.CheckFPAdvSIMDEnabled()
    AArch64.CheckFPEnabled\(\);
    // Check for illegal use of Advanced
    // SIMD in Streaming SVE Mode
    if HaveSME\(\) && PSTATE.SM == '1' && !IsFullA64Enabled\(\) then
        SMEAccessTrap\(SMEExceptionType\_Streaming, PSTATE.EL\);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDTrap

```
// AArch64.CheckFPAdvSIMDTrap()
// =====
// Check against CPTR_EL2 and CPTR_EL3.

AArch64.CheckFPAdvSIMDTrap()
    if HaveEL\(EL3\) && CPTR_EL3.TFP == '1' && EL3SDDUndefPriority\(\) then
        UNDEFINED;

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled\(\) then
        // Check if access disabled in CPTR_EL2
        if HaveVirtHostExt\(\) && HCR_EL2.E2H == '1' then
            boolean disabled;
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap\(EL2\);
        else
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap\(EL2\);

    if HaveEL\(EL3\) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then
            if EL3SDDUndef\(\) then
                UNDEFINED;
            else
                AArch64.AdvSIMDFPAccessTrap\(EL3\);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPEnabled

```
// AArch64.CheckFPEnabled()
// =====
// Check against CPACR[]

AArch64.CheckFPEnabled()
    if PSTATE.EL IN {EL0, EL1} && !IsInHost\(\) then
        // Check if access disabled in CPACR_EL1
        boolean disabled;
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap\(EL1\);

    AArch64.CheckFPAdvSIMDTrap\(\); // Also check against CPTR_EL2 and CPTR_EL3
```



## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForERetTrap

```
// AArch64.CheckForERetTrap()
// =====
// Check for trap on ERET, ERETAA, ERETAB instruction

AArch64.CheckForERetTrap(boolean eret_with_pac, boolean pac_uses_key_a)

    route_to_el2 = FALSE;
    // Non-secure EL1 execution of ERET, ERETAA, ERETAB when either HCR_EL2.NV or
    // HFGITR_EL2.ERET is set, is trapped to EL2
    route_to_el2 = (PSTATE.EL == EL1 && EL2Enabled() &&
        ((HaveNVExt() && HCR_EL2.NV == '1') ||
        (HaveFGTExt() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        HFGITR_EL2.ERET == '1')));
    if route_to_el2 then
        ExceptionRecord except;
        bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;
        except = ExceptionSyndrome(Exception\_ERetTrap);
        if !eret_with_pac then // ERET
            except.syndrome<1> = '0';
            except.syndrome<0> = '0'; // RES0
        else
            except.syndrome<1> = '1';
            if pac_uses_key_a then // ERETAA
                except.syndrome<0> = '0';
            else // ERETAB
                except.syndrome<0> = '1';
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSMCUnDefOrTrap

```
// AArch64.CheckForSMCUnDefOrTrap()
// =====
// Check for UNDEFINED or trap on SMC instruction

AArch64.CheckForSMCUnDefOrTrap(bits(16) imm)
    if PSTATE.EL == EL0 then UNDEFINED;
    if (!(PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1') &&
        HaveEL(EL3) && SCR_EL3.SMD == '1') then
        UNDEFINED;
    route_to_el2 = FALSE;
    if !HaveEL(EL3) then
        if PSTATE.EL == EL1 && EL2Enabled() then
            if HaveNVExt() && HCR_EL2.NV == '1' && HCR_EL2.TSC == '1' then
                route_to_el2 = TRUE;
            else
                UNDEFINED;
        else
            UNDEFINED;
    else
        route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
    if route_to_el2 then
        bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;
        except = ExceptionSyndrome(Exception\_MonitorCall);
        except.syndrome<15:0> = imm;
        except.trappedsyscallinst = TRUE;
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSVCTrap

```
// AArch64.CheckForSVCTrap()
// =====
// Check for trap on SVC instruction

AArch64.CheckForSVCTrap(bits(16) immediate)
  if HaveFGTExt() then
    route_to_el2 = FALSE;
    if PSTATE.EL == EL0 then
      route_to_el2 = (!UsingAArch32() && !ELUsingAArch32(EL1) &&
        EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
        (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));

    elsif PSTATE.EL == EL1 then
      route_to_el2 = (EL2Enabled() && HFGITR_EL2.SVC_EL1 == '1' &&
        (!HaveEL(EL3) || SCR_EL3.FGTEn == '1'));

    if route_to_el2 then
      except = ExceptionSyndrome(Exception_SupervisorCall);
      except.syndrome<15:0> = immediate;
      except.trappedsyscallinst = TRUE;
      bits(64) preferred_exception_return = ThisInstrAddr(64);
      vect_offset = 0x0;

      AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForWFXTrap

```
// AArch64.CheckForWFXTrap()
// =====
// Check for trap on WFE or WFI instruction

AArch64.CheckForWFXTrap(bits(2) target_el, WFXType wfxtype)
  assert HaveEL(target_el);

  boolean is_wfe = wfxtype IN {WFXType_WFE, WFXType_WFET};
  boolean trap;
  case target_el of
    when EL1
      trap = (if is_wfe then SCTLRL_ELx[].nTWE else SCTLRL_ELx[].nTWI) == '0';
    when EL2
      trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
    when EL3
      trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';

  if trap then
    AArch64.WFXTrap(wfxtype, target_el);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckIllegalState

```
// AArch64.CheckIllegalState()
// =====
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch64.CheckIllegalState()
    if PSTATE.IL == '1' then
        route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

        bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;

        except = ExceptionSyndrome(Exception\_IllegalState);

        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
        elsif route_to_el2 then
            AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.MonitorModeTrap

```
// AArch64.MonitorModeTrap()
// =====
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception\_Uncategorized);

    if IsSecureEL2Enabled() then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
        AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.SystemAccessTrap

```
// AArch64.SystemAccessTrap()
// =====
// Trapped access to AArch64 System register or system instruction.

AArch64.SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = AArch64.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.SystemAccessTrapSyndrome

```
// AArch64.SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.

ExceptionRecord AArch64.SystemAccessTrapSyndrome(bits(32) instr_in, integer ec)
    ExceptionRecord except;
    bits(32) instr = instr_in;
    case ec of
        when 0x0 // Trapped access due to unknown reason.
            except = ExceptionSyndrome(ExceptionUncategorized);
        when 0x7 // Trapped access to SVE, Advance SIMD&FP System register.
            except = ExceptionSyndrome(ExceptionAdvSIMDFPAccessTrap);
            except.syndrome<24:20> = ConditionSyndrome();
        when 0x14 // Trapped access to 128-bit System register or
            // 128-bit System instruction.
            except = ExceptionSyndrome(ExceptionSystemRegister128Trap);
            instr = ThisInstr();
            except.syndrome<21:20> = instr<20:19>; // Op0
            except.syndrome<19:17> = instr<7:5>; // Op2
            except.syndrome<16:14> = instr<18:16>; // Op1
            except.syndrome<13:10> = instr<15:12>; // CRn
            except.syndrome<9:6> = instr<4:1>; // Rt
            except.syndrome<4:1> = instr<11:8>; // CRm
            except.syndrome<0> = instr<21>; // Direction
        when 0x18 // Trapped access to System register or system instruction.
            except = ExceptionSyndrome(ExceptionSystemRegisterTrap);
            instr = ThisInstr();
            except.syndrome<21:20> = instr<20:19>; // Op0
            except.syndrome<19:17> = instr<7:5>; // Op2
            except.syndrome<16:14> = instr<18:16>; // Op1
            except.syndrome<13:10> = instr<15:12>; // CRn
            except.syndrome<9:5> = instr<4:0>; // Rt
            except.syndrome<4:1> = instr<11:8>; // CRm
            except.syndrome<0> = instr<21>; // Direction
        when 0x19 // Trapped access to SVE System register
            except = ExceptionSyndrome(ExceptionSVEAccessTrap);
        when 0x1D // Trapped access to SME System register
            except = ExceptionSyndrome(ExceptionSMEAccessTrap);
        otherwise
            Unreachable();

    return except;
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.Undefined

```
// AArch64.Undefined()
// =====

AArch64.Undefined()

    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(ExceptionUncategorized);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elseif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.WFxTrap

```
// AArch64.WFxTrap()
// =====

AArch64.WFxTrap(WFxType wfxtype, bits(2) target_el)
    assert UInt(target_el) > UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception WFxTrap);
    except.syndrome<24:20> = ConditionSyndrome();

    case wfxtype of
        when WFxType_WFI
            except.syndrome<1:0> = '00';
        when WFxType_WFE
            except.syndrome<1:0> = '01';
        when WFxType_WFIT
            except.syndrome<1:0> = '10';
            except.syndrome<2> = '1'; // Register field is valid
            except.syndrome<9:5> = ThisInstr()<4:0>;
        when WFxType_WFET
            except.syndrome<1:0> = '11';
            except.syndrome<2> = '1'; // Register field is valid
            except.syndrome<9:5> = ThisInstr()<4:0>;

    if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64

```
// CheckFPAdvSIMDEnabled64()
// =====
// AArch64 instruction wrapper

CheckFPAdvSIMDEnabled64()
    AArch64.CheckFPAdvSIMDEnabled();
```

## Library pseudocode for aarch64/exceptions/traps/CheckFPEnabled64

```
// CheckFPEnabled64()
// =====
// AArch64 instruction wrapper

CheckFPEnabled64()
    AArch64.CheckFPEnabled();
```

## Library pseudocode for aarch64/exceptions/traps/CheckLDST64BEnabled

```
// CheckLDST64BEnabled()
// =====
// Checks for trap on ST64B and LD64B instructions

CheckLDST64BEnabled()
    boolean trap = FALSE;
    bits(25) iss = ZeroExtend('10', 25); // 0x2
    bits(2) target_el;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnALS == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnALS == '0';
            target_el = EL2;
    else
        target_el = EL1;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnALS == '0';
        target_el = EL2;

    if trap then LDST64BTrap(target_el, iss);
```

## Library pseudocode for aarch64/exceptions/traps/CheckST64BV0Enabled

```
// CheckST64BV0Enabled()
// =====
// Checks for trap on ST64BV0 instruction

CheckST64BV0Enabled()
    boolean trap = FALSE;
    bits(25) iss = ZeroExtend('1', 25); // 0x1
    bits(2) target_el;

    if (PSTATE.EL != EL3 && HaveEL(EL3) &&
        SCR_EL3.EnAS0 == '0' && EL3SDDUndefPriority()) then
        UNDEFINED;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnAS0 == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnAS0 == '0';
            target_el = EL2;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnAS0 == '0';
        target_el = EL2;

    if !trap && PSTATE.EL != EL3 then
        trap = HaveEL(EL3) && SCR_EL3.EnAS0 == '0';
        target_el = EL3;

    if trap then
        if target_el == EL3 && EL3SDDUndef() then
            UNDEFINED;
        else
            LDST64BTrap(target_el, iss);
```

## Library pseudocode for aarch64/exceptions/traps/CheckST64BVEnabled

```
// CheckST64BVEnabled()
// =====
// Checks for trap on ST64BV instruction

CheckST64BVEnabled()
    boolean trap = FALSE;
    bits(25) iss = Zeros(25);
    bits(2) target_el;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnASR == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnASR == '0';
            target_el = EL2;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnASR == '0';
        target_el = EL2;

    if trap then LDST64BTrap(target_el, iss);
```

## Library pseudocode for aarch64/exceptions/traps/LDST64BTrap

```
// LDST64BTrap()
// =====
// Trapped access to LD64B, ST64B, ST64BV and ST64BV0 instructions

LDST64BTrap(bits(2) target_el, bits(25) iss)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception\_LDST64BTrap);
    except.syndrome = iss;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);

    return;
```

## Library pseudocode for aarch64/exceptions/traps/WFETrapDelay

```
// WFETrapDelay()
// =====
// Returns TRUE when delay in trap to WFE is enabled with value to amount of delay,
// FALSE otherwise.

(boolean, integer) WFETrapDelay(bits(2) target_el)
    boolean delay_enabled;
    integer delay;
    case target_el of
        when EL1
            if !IsInHost() then
                delay_enabled = SCTLR_EL1.TWEDEn == '1';
                delay          = 1 << (UInt(SCTLR_EL1.TWEDEL) + 8);
            else
                delay_enabled = SCTLR_EL2.TWEDEn == '1';
                delay          = 1 << (UInt(SCTLR_EL2.TWEDEL) + 8);
        when EL2
            assert EL2Enabled();
            delay_enabled = HCR_EL2.TWEDEn == '1';
            delay          = 1 << (UInt(HCR_EL2.TWEDEL) + 8);
        when EL3
            delay_enabled = SCR_EL3.TWEDEn == '1';
            delay          = 1 << (UInt(SCR_EL3.TWEDEL) + 8);

    return (delay_enabled, delay);
```

## Library pseudocode for aarch64/exceptions/traps/WaitForEventUntilDelay

```
// WaitForEventUntilDelay()
// =====
// Returns TRUE if WaitForEvent() returns before WFE trap delay expires,
// FALSE otherwise.

boolean WaitForEventUntilDelay(boolean delay_enabled, integer delay);
```





```

// AArch64.FaultSyndrome()
// =====
// Creates an exception syndrome value and updates the virtual address for Abort and Watchpoint
// exceptions taken to an Exception level using AArch64.

(bits(25), bits(24)) AArch64.FaultSyndrome(Exception exceptype, FaultRecord fault, boolean pavalid,
                                             bits(64) vaddress)
    assert fault.statuscode != Fault_None;

bits(25) iss = Zeros(25);
bits(24) iss2 = Zeros(24);

boolean d_side = exceptype IN {Exception_DataAbort, Exception_NV2DataAbort,
                               Exception_Watchpoint, Exception_NV2Watchpoint};
if HaveRASExt() && fault.statuscode == Fault_SyncExternal then
    ErrorState errstate = AArch64.PEErrorState(fault);
    iss<12:11> = AArch64.EncodeSyncErrorSyndrome(errstate); // SET

if d_side then
    if fault.accessdesc.acctype == AccessType_GCS then
        iss2<8> = '1';
    if exceptype == Exception_Watchpoint then
        iss<23:0> = WatchpointRelatedSyndrome(fault, vaddress);
    if HaveFeatLS64() && fault.accessdesc.ls64 then
        if (fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_Permission}) then
            (iss2, iss<24:14>) = LS64InstructionSyndrome();
        elsif (IsSecondStage(fault) && !fault.s2fslwalk &&
              (!IsExternalSyncAbort(fault) ||
               (!HaveRASExt() && fault.accessdesc.acctype == AccessType_TTW &&
                boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk")) then
                iss<24:14> = LSInstructionSyndrome();

    if HaveNV2Ext() && fault.accessdesc.acctype == AccessType_NV2 then
        iss<13> = '1'; // Fault is generated by use of VNCR_EL2

    if HaveFeatLS64() && fault.statuscode IN {Fault_AccessFlag, Fault_Translation,
                                              Fault_Permission} then
        iss<12:11> = GetLoadStoreType();

    if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
        iss<8> = '1';

    if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
        iss<6> = '1';
    elsif fault.statuscode IN {Fault_HWUpdateAccessFlag, Fault_Exclusive} then
        iss<6> = bit UNKNOWN;
    elsif fault.accessdesc.atomicop && IsExternalAbort(fault) then
        iss<6> = bit UNKNOWN;
    else
        iss<6> = if fault.write then '1' else '0';

    if fault.statuscode == Fault_Permission then
        iss2<5> = if fault.dirtybit then '1' else '0';
        iss2<6> = if fault.overlay then '1' else '0';
        if iss<24> == '0' then
            iss<21> = if fault.toplevel then '1' else '0';
            iss2<7> = if fault.assuredonly then '1' else '0';
            iss2<9> = if fault.tagaccess then '1' else '0';
            iss2<10> = if fault.s1tagnotdata then '1' else '0';

else
    if (fault.accessdesc.acctype == AccessType_IFETCH &&
        fault.statuscode == Fault_Permission) then
        iss2<5> = if fault.dirtybit then '1' else '0';
        iss<21> = if fault.toplevel then '1' else '0';
        iss2<7> = if fault.assuredonly then '1' else '0';
        iss2<6> = if fault.overlay then '1' else '0';
    if IsExternalAbort(fault) then iss<9> = fault.extflag;
    iss<7> = if fault.s2fslwalk then '1' else '0';
    iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

```

```
return (iss, iss2);
```

### Library pseudocode for aarch64/functions/aborts/EncodeGPCSC

```
// EncodeGPCSC()
// =====
// Function that gives the GPCSC code for types of GPT Fault

bits(6) EncodeGPCSC(GPCFRecord gpcf)
    assert gpcf.level IN {0,1};

    case gpcf.gpf of
        when GPCF AddressSize return '00000':gpcf.level<0>;
        when GPCF Walk         return '00010':gpcf.level<0>;
        when GPCF Fail         return '00110':gpcf.level<0>;
        when GPCF EABT         return '01010':gpcf.level<0>;
```

### Library pseudocode for aarch64/functions/aborts/LS64InstructionSyndrome

```
// LS64InstructionSyndrome()
// =====
// Returns the syndrome information and LST for a Data Abort by a
// ST64B, ST64BV, ST64BV0, or LD64B instruction. The syndrome information
// includes the ISS2, extended syndrome field.

(bits(24), bits(11)) LS64InstructionSyndrome();
```

### Library pseudocode for aarch64/functions/aborts/WatchpointFARNotPrecise

```
// WatchpointFARNotPrecise()
// =====
// Returns TRUE If the lowest watchpointed address that is higher than or equal to the address
// recorded in EDWAR might not have been accessed by the instruction, other than the CONSTRAINED
// UNPREDICTABLE condition of watchpoint matching a range of addresses with lowest address 16 bytes
// rounded down and upper address rounded up to nearest 16 byte multiple,
// FALSE otherwise.

boolean WatchpointFARNotPrecise(FaultRecord fault);
```



```

// AArch64.AT()
// =====
// Perform address translation as per AT instructions.

AArch64.AT(bits(64) address, TranslationStage stage_in, bits(2) el_in, ATAccess ataccess)
    TranslationStage stage = stage_in;
    bits(2) el = el_in;
    bits(2) effective_nse_ns = EffectiveSCR\_EL3\_NSE() : EffectiveSCR\_EL3\_NS();
    if HaveRME() && PSTATE.EL == EL3 && effective_nse_ns == '10' && el != EL3 then
        UNDEFINED;
    // For stage 1 translation, when HCR_EL2.{E2H, TGE} is {1,1} and requested EL is EL1,
    // the EL2&0 translation regime is used.
    if EL2Enabled() && HCR_EL2.<E2H, TGE> == '11' && el == EL1 && stage == TranslationStage\_1 then
        el = EL2;
    if HaveEL(EL3) && stage == TranslationStage\_12 && !EL2Enabled() then
        stage = TranslationStage\_1;

    SecurityState ss = SecurityStateAtEL(el);

    accdesc = CreateAccDescAT(ss, el, ataccess);
    aligned = TRUE;

    FaultRecord fault = NoFault(accdesc);
    Regime regime;
    if stage == TranslationStage\_12 then
        regime = Regime\_EL10;
    else
        regime = TranslationRegime(el);

    AddressDescriptor addrdesc;
    if (el == EL0 && ELUsingAArch32(EL1)) || (el != EL0 && ELUsingAArch32(el)) then
        if regime == Regime\_EL2 || TTBCR.EAE == '1' then
            (fault, addrdesc) = AArch32.S1TranslateLD(fault, regime, address<31:0>, aligned, accdesc);
        else
            (fault, addrdesc, -) = AArch32.S1TranslateSD(fault, regime, address<31:0>, aligned, accdesc);
    else
        (fault, addrdesc) = AArch64.S1Translate(fault, regime, address, aligned, accdesc);

    if stage == TranslationStage\_12 && fault.statuscode == Fault\_None then
        boolean slaarch64;
        if ELUsingAArch32(EL1) && regime == Regime\_EL10 && EL2Enabled() then
            addrdesc.vaddress = ZeroExtend(address, 64);
            (fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, aligned, accdesc);
        elsif regime == Regime\_EL10 && EL2Enabled() then
            slaarch64 = TRUE;
            (fault, addrdesc) = AArch64.S2Translate(fault, addrdesc, slaarch64, aligned, accdesc);

    is_ATS1Ex = stage != TranslationStage\_12;
    if fault.statuscode != Fault\_None then
        addrdesc = CreateFaultyAddressDescriptor(address, fault);
        // Take an exception on:
        // * A Synchronous External abort occurs on translation table walk
        // * A stage 2 fault occurs on a stage 1 walk
        // * A GPC Exception (FEAT_RME)
        // * A GPF from ATS1E{1,0}* when executed from EL1 and HCR_EL2.GPF == '1' (FEAT_RME)
        if (IsExternalAbort(fault) ||
            (PSTATE.EL == EL1 && fault.s2fslwalk) ||
            (HaveRME() && fault.gpcf.gpf != GPCF\_None && (
                ReportAsGPCEXception(fault) ||
                (EL2Enabled() && HCR_EL2.GPF == '1' && PSTATE.EL == EL1 && el IN {EL1, EL0} &&
                    is_ATS1Ex)
            ))) then
            PAR_EL1 = bits(128) UNKNOWN;
            AArch64.Abort(address, addrdesc.fault);

    AArch64.EncodePAR(regime, is_ATS1Ex, addrdesc);
    return;

```

## Library pseudocode for aarch64/functions/at/AArch64.EncodePAR

```
// AArch64.EncodePAR()
// =====
// Encode PAR register with result of translation.

AArch64.EncodePAR(Regime regime, boolean is_ATS1Ex, AddressDescriptor addrdesc)
    PAR_EL1 = Zeros(128);
    paspace = addrdesc.paddress.paspace;

    if AArch64.isPARFormatD128(regime, is_ATS1Ex) then
        PAR_EL1.D128 = '1';
    else
        PAR_EL1.D128 = '0';

    if !IsFault(addrdesc) then
        PAR_EL1.F = '0';
        if HaveRME() then
            if regime == Regime\_EL3 then
                case paspace of
                    when PAS\_Secure      PAR_EL1.<NSE,NS> = '00';
                    when PAS\_NonSecure PAR_EL1.<NSE,NS> = '01';
                    when PAS\_Root      PAR_EL1.<NSE,NS> = '10';
                    when PAS\_Realm    PAR_EL1.<NSE,NS> = '11';

            elsif SecurityStateForRegime(regime) == SS\_Secure then
                PAR_EL1.NSE = bit UNKNOWN;
                PAR_EL1.NS = if paspace == PAS\_Secure then '0' else '1';

            elsif SecurityStateForRegime(regime) == SS\_Realm then
                if regime == Regime\_EL10 && is_ATS1Ex then
                    PAR_EL1.NSE = bit UNKNOWN;
                    PAR_EL1.NS = bit UNKNOWN;
                else
                    PAR_EL1.NSE = bit UNKNOWN;
                    PAR_EL1.NS = if paspace == PAS\_Realm then '0' else '1';

            else
                PAR_EL1.NSE = bit UNKNOWN;
                PAR_EL1.NS = bit UNKNOWN;
        else
            PAR_EL1<11> = '1'; // RES1
            if SecurityStateForRegime(regime) == SS\_Secure then
                PAR_EL1.NS = if paspace == PAS\_Secure then '0' else '1';
            else
                PAR_EL1.NS = bit UNKNOWN;
        PAR_EL1.SH = ReportedPARShareability(PAREncodeShareability(addrdesc.memattrs));
        if PAR_EL1.D128 == '1' then
            PAR_EL1<119:76> = addrdesc.paddress.address<55:12>;
        else
            PAR_EL1<55:12> = addrdesc.paddress.address<55:12>;
        PAR_EL1.ATTR = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
        PAR_EL1<10> = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";
    else
        PAR_EL1.F = '1';
        PAR_EL1.DirtyBit = if addrdesc.fault.dirtybit then '1' else '0';
        PAR_EL1.Overlay = if addrdesc.fault.overlay then '1' else '0';
        PAR_EL1.TopLevel = if addrdesc.fault.toplevel then '1' else '0';
        PAR_EL1.AssuredOnly = if addrdesc.fault.assuredonly then '1' else '0';
        PAR_EL1.FST = AArch64.PARFaultStatus(addrdesc.fault);
        PAR_EL1.PTW = if addrdesc.fault.s2fslwalk then '1' else '0';
        PAR_EL1.S = if addrdesc.fault.secondstage then '1' else '0';
        PAR_EL1<11> = '1'; // RES1
        PAR_EL1<63:48> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";
    return;
```

## Library pseudocode for aarch64/functions/at/AArch64.PARFaultStatus

```
// AArch64.PARFaultStatus()
// =====
// Fault status field decoding of 64-bit PAR.

bits(6) AArch64.PARFaultStatus(FaultRecord fault)
    bits(6) fst;

    if fault.statuscode == Fault\_Domain then
        // Report Domain fault
        assert fault.level IN {1,2};
        fst<1:0> = if fault.level == 1 then '01' else '10';
        fst<5:2> = '1111';
    else
        fst = EncodeLDFSC(fault.statuscode, fault.level);
    return fst;
```

## Library pseudocode for aarch64/functions/at/AArch64.isPARFormatD128

```
// AArch64.isPARFormatD128()
// =====
// Check if last stage of translation uses VMSAv9-128.
// Last stage of translation is stage 2 if enabled, else it is stage 1.

boolean AArch64.isPARFormatD128(Regime regime, boolean is_ATS1Ex)
    boolean isPARFormatD128;
    // Regime_EL2 does not support VMSAv9-128
    if regime == Regime\_EL2 || !Have128BitDescriptorExt() then
        isPARFormatD128 = FALSE;
    else
        isPARFormatD128 = FALSE;
        case regime of
            when Regime\_EL3
                isPARFormatD128 = TCR_EL3.D128 == '1';
            when Regime\_EL20
                isPARFormatD128 = TCR2_EL2.D128 == '1';
            when Regime\_EL10
                if is_ATS1Ex || !EL2Enabled() || HCR_EL2.<VM,DC> == '00' then
                    isPARFormatD128 = TCR2_EL1.D128 == '1';
                else
                    isPARFormatD128 = VTCR_EL2.D128 == '1';

    return isPARFormatD128;
```

## Library pseudocode for aarch64/functions/at/GetPAR\_EL1\_D128

```
// GetPAR_EL1_D128()
// =====
// Query the PAR_EL1.D128 field

bit GetPAR_EL1_D128()
    bit D128;

    D128 = PAR_EL1.D128;
    return D128;
```

## Library pseudocode for aarch64/functions/at/GetPAR\_EL1\_F

```
// GetPAR_EL1_F()
// =====
// Query the PAR_EL1.F field.

bit GetPAR_EL1_F()
    bit F;

    F = PAR_EL1.F;
    return F;
```

## Library pseudocode for aarch64/functions/barrierop/MemBarrierOp

```
// MemBarrierOp
// =====
// Memory barrier instruction types.

enumeration MemBarrierOp { MemBarrierOp_DSB          // Data Synchronization Barrier
                           , MemBarrierOp_DMB          // Data Memory Barrier
                           , MemBarrierOp_ISB          // Instruction Synchronization Barrier
                           , MemBarrierOp_SSBB         // Speculative Synchronization Barrier to VA
                           , MemBarrierOp_PSSBB        // Speculative Synchronization Barrier to PA
                           , MemBarrierOp_SB           // Speculation Barrier
                           };
```

## Library pseudocode for aarch64/functions/bfxpreferred/BFXPreferred

```
// BFXPreferred()
// =====
//
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

boolean BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)

    // must not match UBFIZ/SBFIX alias
    if UInt(imms) < UInt(immr) then
        return FALSE;

    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf:'11111' then
        return FALSE;

    // must not match UXTx/SXTx alias
    if immr == '000000' then
        // must not match 32-bit UXT[BH] or SXT[BH]
        if sf == '0' && imms IN {'000111', '001111'} then
            return FALSE;
        // must not match 64-bit SXT[BHW]
        if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then
            return FALSE;

    // must be UBFX/SBFX alias
    return TRUE;
```





```

// AltDecodeBitMasks()
// =====
// Alternative but logically equivalent implementation of DecodeBitMasks() that
// uses simpler primitives to compute tmask and wmask.

(bits(M), bits(M)) AltDecodeBitMasks(bit immN, bits(6) imms, bits(6) immr,
                                     boolean immediate, integer M)

    bits(64) tmask, wmask;
    bits(6) tmask_and, wmask_and;
    bits(6) tmask_or, wmask_or;
    bits(6) levels;

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    len = HighestSetBit(immN:NOT(imms));
    if len < 1 then UNDEFINED;
    assert M >= (1 << len);

    // Determine s, r and s - r parameters
    levels = ZeroExtend(Ones(len), 6);

    // For logical immediates an all-ones value of s is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then
        UNDEFINED;

    s = UInt(imms AND levels);
    r = UInt(immr AND levels);
    diff = s - r;    // 6-bit subtract with borrow

    // Compute "top mask"
    tmask_and = diff<5:0> OR NOT(levels);
    tmask_or  = diff<5:0> AND levels;

    tmask = Ones(64);
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<0>, 1) : Ones(1), 32))
        OR Replicate(Zeros(1) : Replicate(tmask_or<0>, 1), 32));
    // optimization of first step:
    // tmask = Replicate(tmask_and<0> : '1', 32);
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<1>, 2) : Ones(2), 16))
        OR Replicate(Zeros(2) : Replicate(tmask_or<1>, 2), 16));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<2>, 4) : Ones(4), 8))
        OR Replicate(Zeros(4) : Replicate(tmask_or<2>, 4), 8));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<3>, 8) : Ones(8), 4))
        OR Replicate(Zeros(8) : Replicate(tmask_or<3>, 8), 4));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<4>, 16) : Ones(16), 2))
        OR Replicate(Zeros(16) : Replicate(tmask_or<4>, 16), 2));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<5>, 32) : Ones(32), 1))
        OR Replicate(Zeros(32) : Replicate(tmask_or<5>, 32), 1));

    // Compute "wraparound mask"
    wmask_and = immr OR NOT(levels);
    wmask_or  = immr AND levels;

    wmask = Zeros(64);
    wmask = ((wmask
        AND Replicate(Ones(1) : Replicate(wmask_and<0>, 1), 32))
        OR Replicate(Replicate(wmask_or<0>, 1) : Zeros(1), 32));
    // optimization of first step:
    // wmask = Replicate(wmask_or<0> : '0', 32);
    wmask = ((wmask
        AND Replicate(Ones(2) : Replicate(wmask_and<1>, 2), 16))
        OR Replicate(Replicate(wmask_or<1>, 2) : Zeros(2), 16));
    wmask = ((wmask

```

```

        AND Replicate(Ones(4) : Replicate(wmask_and<2>, 4), 8))
        OR Replicate(Replicate(wmask_or<2>, 4) : Zeros(4), 8));
wmask = ((wmask
        AND Replicate(Ones(8) : Replicate(wmask_and<3>, 8), 4))
        OR Replicate(Replicate(wmask_or<3>, 8) : Zeros(8), 4));
wmask = ((wmask
        AND Replicate(Ones(16) : Replicate(wmask_and<4>, 16), 2))
        OR Replicate(Replicate(wmask_or<4>, 16) : Zeros(16), 2));
wmask = ((wmask
        AND Replicate(Ones(32) : Replicate(wmask_and<5>, 32), 1))
        OR Replicate(Replicate(wmask_or<5>, 32) : Zeros(32), 1));

if diff<6> != '0' then // borrow from s - r
    wmask = wmask AND tmask;
else
    wmask = wmask OR tmask;

return (wmask<M-1:0>, tmask<M-1:0>);

```

## Library pseudocode for aarch64/functions/bitmasks/DecodeBitMasks

```

// DecodeBitMasks()
// =====
// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

(bits(M), bits(M)) DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr,
                                   boolean immediate, integer M)

    bits(M) tmask, wmask;
    bits(6) levels;

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    len = HighestSetBit(immN:NOT(imms));
    if len < 1 then UNDEFINED;
    assert M >= (1 << len);

    // Determine s, r and s - r parameters
    levels = ZeroExtend(Ones(len), 6);

    // For logical immediates an all-ones value of s is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then
        UNDEFINED;

    s = UInt(imms AND levels);
    r = UInt(immr AND levels);
    diff = s - r;    // 6-bit subtract with borrow

    esize = 1 << len;
    d = UInt(diff<len-1:0>);
    welem = ZeroExtend(Ones(s + 1), esize);
    telem = ZeroExtend(Ones(d + 1), esize);
    wmask = Replicate(ROR(welem, r), M DIV esize);
    tmask = Replicate(telem, M DIV esize);
    return (wmask, tmask);

```

## Library pseudocode for aarch64/functions/cache/AArch64.DataMemZero

```
// AArch64.DataMemZero()
// =====
// Write Zero to data memory.

AArch64.DataMemZero(bits(64) regval, bits(64) vaddress, AccessDescriptor accdesc_in, integer size)
AccessDescriptor accdesc = accdesc_in;

// If the instruction targets tags as a payload, confer with system register configuration
// which may override this.
if HaveMTE2Ext() && accdesc.tagaccess then
    accdesc.tagaccess = AArch64.AllocationTagAccessIsEnabled(accdesc.el);

// If the instruction encoding permits tag checking, confer with system register configuration
// which may override this.
if HaveMTE2Ext() && accdesc.tagchecked then
    accdesc.tagchecked = AArch64.AccessIsTagChecked(vaddress, accdesc);

boolean aligned = TRUE;
AddressDescriptor memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);

if IsFault(memaddrdesc) then
    if IsDebugException(memaddrdesc.fault) then
        AArch64.Abort(vaddress, memaddrdesc.fault);
    else
        AArch64.Abort(regval, memaddrdesc.fault);

if HaveTME() then
    if accdesc.transactional && !MemHasTransactionalAccess(memaddrdesc.memattrs) then
        FailTransaction(TMFailure\_IMP, FALSE);

for i = 0 to size-1
    if HaveMTE2Ext() && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(vaddress);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            if (boolean IMPLEMENTATION_DEFINED
                "DC_ZVA tag fault reported with lowest faulting address") then
                AArch64.TagCheckFault(vaddress, accdesc);
            else
                AArch64.TagCheckFault(regval, accdesc);
        memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, Zeros(8));
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
return;
```

## Library pseudocode for aarch64/functions/cache/AArch64.TagMemZero

```
// AArch64.TagMemZero()
// =====
// Write Zero to tag memory.

AArch64.TagMemZero(bits(64) regval, bits(64) vaddress, AccessDescriptor accdesc_in, integer size)
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

    AccessDescriptor accdesc = accdesc_in;

    integer count = size >> LOG2\_TAG\_GRANULE;
    bits(4) tag = AArch64.AllocationTagFromAddress(vaddress);
    boolean aligned = IsAligned(vaddress, TAG\_GRANULE);

    // Stores of allocation tags must be aligned
    if !aligned then
        AArch64.Abort(vaddress, AlignmentFault(accdesc));

    if HaveMTE2Ext() then
        accdesc.tagaccess = AArch64.AllocationTagAccessIsEnabled(accdesc.el);

    memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        if IsDebugException(memaddrdesc.fault) then
            AArch64.Abort(vaddress, memaddrdesc.fault);
        else
            AArch64.Abort(regval, memaddrdesc.fault);

    if !accdesc.tagaccess || memaddrdesc.memattrs.tags != MemTag\_AllocationTagged then
        return;

    for i = 0 to count-1
        memstatus = PhysMemTagWrite(memaddrdesc, accdesc, tag);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + TAG\_GRANULE;

    return;
```

## Library pseudocode for aarch64/functions/compareop/CompareOp

```
// CompareOp
// =====
// Vector compare instruction types.

enumeration CompareOp    {CompareOp_GT, CompareOp_GE, CompareOp_EQ,
                          CompareOp_LE, CompareOp_LT};
```

## Library pseudocode for aarch64/functions/countop/CountOp

```
// CountOp
// =====
// Bit counting instruction types.

enumeration CountOp      {CountOp_CLZ, CountOp_CLS, CountOp_CNT};
```

## Library pseudocode for aarch64/functions/d128/IsD128Enabled

```
// IsD128Enabled()
// =====
// Returns true if 128-bit page descriptor is enabled

boolean IsD128Enabled(bits(2) el)
    boolean d128enabled;
    if Have128BitDescriptorExt() then
        case el of
            when EL0
                if !ELIsInHost(EL0) then
                    d128enabled = IsTCR2EL1Enabled() && TCR2_EL1.D128 == '1';
                else
                    d128enabled = IsTCR2EL2Enabled() && TCR2_EL2.D128 == '1';
            when EL1
                d128enabled = IsTCR2EL1Enabled() && TCR2_EL1.D128 == '1';
            when EL2
                d128enabled = IsTCR2EL2Enabled() && HCR_EL2.E2H == '1' && TCR2_EL2.D128 == '1';
            when EL3
                d128enabled = TCR_EL3.D128 == '1';
        else
            d128enabled = FALSE;

    return d128enabled;
```



```

// AArch64.DC()
// =====
// Perform Data Cache Operation.

AArch64.DC(bits(64) regval, CacheType cachetype, CacheOp cacheop, CacheOpScope opscope_in)
CacheOpScope opscope = opscope_in;
CacheRecord cache;

cache.acctype = AccessType\_DC;
cache.cachetype = cachetype;
cache.cacheop = cacheop;
cache.opscope = opscope;

if opscope == CacheOpScope\_SetWay then
    ss = SecurityStateAtEL(PSTATE.EL);
    cache.cpas = CPASAtSecurityState(ss);
    cache.shareability = Shareability\_NSH;
    (cache.setnum, cache.waynum, cache.level) = DecodeSW(regval, cachetype);
    if (cacheop == CacheOp\_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
        (HCR_EL2.SWIO == '1' || HCR_EL2.<DC,VM> != '00')) then
        cache.cacheop = CacheOp\_CleanInvalidate;

    CACHE\_OP(cache);
    return;

if EL2Enabled() && !IsInHost() then
    if PSTATE.EL IN {EL0, EL1} then
        cache.is_vmid_valid = TRUE;
        cache.vmid = VMID[];
    else
        cache.is_vmid_valid = FALSE;
else
    cache.is_vmid_valid = FALSE;

if PSTATE.EL == EL0 then
    cache.is_asid_valid = TRUE;
    cache.asid = ASID[];
else
    cache.is_asid_valid = FALSE;

if (opscope == CacheOpScope\_PoDP &&
    boolean IMPLEMENTATION_DEFINED "Memory system does not supports PoDP") then
    opscope = CacheOpScope\_PoP;
if (opscope == CacheOpScope\_PoP &&
    boolean IMPLEMENTATION_DEFINED "Memory system does not supports PoP") then
    opscope = CacheOpScope\_PoC;
vaddress = regval;

size = 0; // by default no watchpoint address
if cacheop == CacheOp\_Invalidate then
    size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
    assert size >= 4*(2^(UInt(CTR_EL0.DminLine))) && size <= 2048;
    assert UInt(size<32:0> AND (size-1)<32:0>) == 0; // size is power of 2
    vaddress = Align(regval, size);

if DCInstNeedsTranslation(opscope) then
    cache.vaddress = vaddress;
    boolean aligned = TRUE;
    AccessDescriptor accdesc = CreateAccDescDC(cache);
    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);
    if IsFault(memaddrdesc) then
        AArch64.Abort(regval, memaddrdesc.fault);

    cache.translated = TRUE;
    cache.paddress = memaddrdesc.paddress;
    cache.cpas = CPASAtPAS(memaddrdesc.paddress.paspace);
    if opscope IN {CacheOpScope\_PoC, CacheOpScope\_PoP, CacheOpScope\_PoDP} then
        cache.shareability = memaddrdesc.memattrs.shareability;
    else
        cache.shareability = Shareability\_NSH;

```



```

elseif opscope == CacheOpScope\_PoE then
    cache.translated      = TRUE;
    cache.shareability    = Shareability\_OSH;
    cache.paddress.address = regval<55:0>;
    cache.paddress.paspace = DecodePASpace(regval<62>, regval<63>);
    cache.cpas            = CPASAtPAS(cache.paddress.paspace);

    // If a Reserved encoding is selected, the instruction is permitted to be treated as a NOP.
    if cache.paddress.paspace != PAS\_Realm then
        EndOfInstruction();

    if boolean IMPLEMENTATION_DEFINED "Apply granule protection check on DC to PoE" then
        AddressDescriptor memaddrdesc;
        AccessDescriptor accdesc = CreateAccDescDC(cache);
        memaddrdesc.paddress      = cache.paddress;
        memaddrdesc.fault.gpcf    = GranuleProtectionCheck(memaddrdesc, accdesc);

        if memaddrdesc.fault.gpcf.gpf != GPCF\_None then
            memaddrdesc.fault.statuscode = Fault\_GPCFOnOutput;
            memaddrdesc.fault.paddress   = memaddrdesc.paddress;
            AArch64.Abort(bits(64) UNKNOWN, memaddrdesc.fault);
    elseif opscope == CacheOpScope\_PoPA then
        cache.translated      = TRUE;
        cache.shareability    = Shareability\_OSH;
        cache.paddress.address = regval<55:0>;
        cache.paddress.paspace = DecodePASpace(regval<62>, regval<63>);
        cache.cpas            = CPASAtPAS(cache.paddress.paspace);
    else
        cache.vaddress        = vaddress;
        cache.translated      = FALSE;
        cache.shareability    = Shareability UNKNOWN;
        cache.paddress        = FullAddress UNKNOWN;

    if (cacheop == CacheOp\_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
        HCR_EL2.<DC,VM> != '00') then
        cache.cacheop = CacheOp\_CleanInvalidate;

    // If Secure state is not implemented, but RME is, the instruction acts as a NOP
    if cache.translated && cache.cpas == CPAS\_Secure && !HaveSecureState() then
        return;

    CACHE\_OP(cache);
    return;

```

## Library pseudocode for aarch64/functions/dc/AArch64.MemZero

```

// AArch64.MemZero()
// =====

AArch64.MemZero(bits(64) regval, CacheType cachetype)
    integer size = 4*(2^(UInt(DCZID_EL0.BS)));
    assert size <= MAX\_ZERO\_BLOCK\_SIZE;
    if HaveMTE2Ext() then
        assert size >= TAG\_GRANULE;

    bits(64) vaddress = Align(regval, size);

    boolean tagaccess = cachetype IN {CacheType\_Tag, CacheType\_Data\_Tag};
    boolean tagchecked = cachetype == CacheType\_Data;
    AccessDescriptor accdesc = CreateAccDescDCZero(tagaccess, tagchecked);

    if cachetype IN {CacheType\_Tag, CacheType\_Data\_Tag} then
        AArch64.TagMemZero(regval, vaddress, accdesc, size);

    if cachetype IN {CacheType\_Data, CacheType\_Data\_Tag} then
        AArch64.DataMemZero(regval, vaddress, accdesc, size);

    return;

```

## Library pseudocode for aarch64/functions/dc/MemZero

```
constant integer MAX_ZERO_BLOCK_SIZE = 2048;
```

## Library pseudocode for aarch64/functions/eret/AArch64.ExceptionReturn

```
// AArch64.ExceptionReturn()
// =====

AArch64.ExceptionReturn(bits(64) new_pc_in, bits(64) spsr)
    bits(64) new_pc = new_pc_in;
    if HaveTME() && TSTATE.depth > 0 then
        FailTransaction(TMFailure_ERR, FALSE);

    if HaveIESB() then
        sync_errors = SCTLRL_ELx[].IESB == '1';
        if HaveDoubleFaultExt() then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && PSTATE.EL == EL3);
        if sync_errors then
            SynchronizeErrors();
            iesb_req = TRUE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

    boolean brbe_source_allowed = FALSE;
    bits(64) brbe_source_address = Zeros(64);
    if HaveBRBExt() then
        brbe_source_allowed = BranchRecordAllowed(PSTATE.EL);
        brbe_source_address = PC[];

    if !IsFeatureImplemented(FEAT_ExS) || SCTLRL_ELx[].EOS == '1' then
        SynchronizeContext();

    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    bits(2) source_el = PSTATE.EL;
    boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    SetPSTATEFromPSR(spsr, illegal_psr_state);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if illegal_psr_state && spsr<4> == '1' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    elseif UsingAArch32() then // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the
        // target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0'; // T32
        else
            new_pc<1:0> = '00'; // A32
    else // Return to AArch64
        // ELR_ELx[63:56] might include a tag
        new_pc = AArch64.BranchAddr(new_pc, PSTATE.EL);

    if HaveBRBExt() then
        BRBEEExceptionReturn(new_pc, source_el,
            brbe_source_allowed, brbe_source_address);

    if UsingAArch32() then
        if HaveSME() && PSTATE.SM == '1' then ResetSVEState();

        // 32 most significant bits are ignored.
        boolean branch_conditional = FALSE;
        BranchTo(new_pc<31:0>, BranchType_ERET, branch_conditional);
    else
        BranchToAddr(new_pc, BranchType_ERET);

    CheckExceptionCatch(FALSE); // Check for debug event on exception return
```

### Library pseudocode for aarch64/functions/exclusive/AArch64.ExclusiveMonitorsPass

```
// AArch64.ExclusiveMonitorsPass()
// =====
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

// It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
// before or after the check on the local Exclusives monitor. As a result a failure
// of the local monitor can occur on some implementations even if the memory
// access would give an memory abort.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size, AccessDescriptor accdesc)
    boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    if !AArch64.IsExclusiveVA(address, ProcessorID(), size) then
        return FALSE;

    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.address, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed && memaddrdesc.memattrs.shareability != Shareability\_NSH then
        passed = IsExclusiveGlobal(memaddrdesc.address, ProcessorID(), size);

    return passed;
```

### Library pseudocode for aarch64/functions/exclusive/AArch64.IsExclusiveVA

```
// AArch64.IsExclusiveVA()
// =====
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.

boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);
```

### Library pseudocode for aarch64/functions/exclusive/AArch64.MarkExclusiveVA

```
// AArch64.MarkExclusiveVA()
// =====
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.

AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);
```

## Library pseudocode for aarch64/functions/exclusive/AArch64.SetExclusiveMonitors

```
// AArch64.SetExclusiveMonitors()
// =====
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch64.SetExclusiveMonitors(bits(64) address, integer size)
    boolean acqrel = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescExLDST(MemOp\_LOAD, acqrel, tagchecked);
    boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

        MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch64.MarkExclusiveVA(address, ProcessorID(), size);
```

## Library pseudocode for aarch64/functions/extendreg/DecodeRegExtend

```
// DecodeRegExtend()
// =====
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)
    case op of
        when '000' return ExtendType\_UXTB;
        when '001' return ExtendType\_UXTH;
        when '010' return ExtendType\_UXTW;
        when '011' return ExtendType\_UXTX;
        when '100' return ExtendType\_SXTB;
        when '101' return ExtendType\_SXTH;
        when '110' return ExtendType\_SXTW;
        when '111' return ExtendType\_SXTX;
```

## Library pseudocode for aarch64/functions/extendreg/ExtendReg

```
// ExtendReg()
// =====
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType exttype, integer shift, integer N)
    assert shift >= 0 && shift <= 4;
    bits(N) val = X[reg, N];
    boolean unsigned;
    integer len;

    case exttype of
        when ExtendType\_SXTB unsigned = FALSE; len = 8;
        when ExtendType\_SXTH unsigned = FALSE; len = 16;
        when ExtendType\_SXTW unsigned = FALSE; len = 32;
        when ExtendType\_SCTX unsigned = FALSE; len = 64;
        when ExtendType\_UXTB unsigned = TRUE; len = 8;
        when ExtendType\_UXTH unsigned = TRUE; len = 16;
        when ExtendType\_UXTW unsigned = TRUE; len = 32;
        when ExtendType\_UCTX unsigned = TRUE; len = 64;

    // Note the extended width of the intermediate value and
    // that sign extension occurs from bit <len+shift-1>, not
    // from bit <len-1>. This is equivalent to the instruction
    // [SU]BFIZ Rtmp, Rreg, #shift, #len
    // It may also be seen as a sign/zero extend followed by a shift:
    // LSL(Extend(val<len-1:0>, N, unsigned), shift);

    len = Min(len, N - shift);
    return Extend(val<len-1:0> : Zeros(shift), N, unsigned);
```

## Library pseudocode for aarch64/functions/extendreg/ExtendType

```
// ExtendType
// =====
// AArch64 register extend and shift.

enumeration ExtendType {ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW, ExtendType_SCTX,
                        ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UCTX};
```

## Library pseudocode for aarch64/functions/fpconvop/FPCConvOp

```
// FPCConvOp
// =====
// Floating-point convert/move instruction types.

enumeration FPCConvOp {FPCConvOp_CVT_FtoI, FPCConvOp_CVT_ItoF,
                      FPCConvOp_MOV_FtoI, FPCConvOp_MOV_ItoF,
                      , FPCConvOp_CVT_FtoI_JS
};
```

## Library pseudocode for aarch64/functions/fpmaxminop/FPMaxMinOp

```
// FPMaxMinOp
// =====
// Floating-point min/max instruction types.

enumeration FPMaxMinOp {FPMaxMinOp_MAX, FPMaxMinOp_MIN,
                      FPMaxMinOp_MAXNUM, FPMaxMinOp_MINNUM};
```

## Library pseudocode for aarch64/functions/fpunaryop/FPUnaryOp

```
// FPUnaryOp
// =====
// Floating-point unary instruction types.

enumeration FPUnaryOp    {FPUnaryOp_ABS, FPUnaryOp_MOV,
                          FPUnaryOp_NEG, FPUnaryOp_SQRT};
```

## Library pseudocode for aarch64/functions/fusedrstep/FPRSqrtStepFused

```
// FPRSqrtStepFused()
// =====

bits(N) FPRSqrtStepFused(bits(N) op1_in, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) result;
    bits(N) op1 = op1_in;
    boolean done;
    FPCRTType fpcr = FPCR[];
    op1 = FPNeg(op1);
    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    boolean fpexc = !altfp; // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00'; // Use RNE rounding mode

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    FPRounding rounding = FPRoundingMode(fpcr);

    if !done then
        inf1 = (type1 == FPType\_Infinity);
        inf2 = (type2 == FPType\_Infinity);
        zero1 = (type1 == FPType\_Zero);
        zero2 = (type2 == FPType\_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPOnePointFive('0', N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        else
            // Fully fused multiply-add and halve
            result_value = (3.0 + (value1 * value2)) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

    return result;
```

## Library pseudocode for aarch64/functions/fusedrstep/FPRecipStepFused

```
// FPRecipStepFused()
// =====

bits(N) FPRecipStepFused(bits(N) op1_in, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) op1 = op1_in;
    bits(N) result;
    boolean done;
    FPCRType fpcr = FPCR\[\];
    op1 = FPNeg(op1);

    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    boolean fpexc = !altfp; // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00'; // Use RNE rounding mode

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    FPRounding rounding = FPRoundingMode(fpcr);

    if !done then
        inf1 = (type1 == FPTType\_Infinity);
        inf2 = (type2 == FPTType\_Infinity);
        zero1 = (type1 == FPTType\_Zero);
        zero2 = (type2 == FPTType\_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo('0', N);
        elsif inf1 || inf2 then
            result = FPIInfinity(sign1 EOR sign2, N);
        else
            // Fully fused multiply-add
            result_value = 2.0 + (value1 * value2);
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

    return result;
```

## Library pseudocode for aarch64/functions/gcs/AddGCSExRecord

```
// AddGCSExRecord()
// =====
// Generates and then writes an exception record to the
// current Guarded control stack.

AddGCSExRecord(bits(64) elr, bits(64) spsr, bits(64) lr)
    bits(64) ptr;
    AccessDescriptor accdesc = CreateAccDescGCS(PSTATE.EL, MemOp\_STORE);

    ptr = GetCurrentGCSPtr();

    // Store the record
    Mem[ptr-8, 8, accdesc] = lr;
    Mem[ptr-16, 8, accdesc] = spsr;
    Mem[ptr-24, 8, accdesc] = elr;
    Mem[ptr-32, 8, accdesc] = Zeros(60):'1001';

    // Decrement the pointer value
    ptr = ptr - 32;

    SetCurrentGCSPtr(ptr);
    return;
```

## Library pseudocode for aarch64/functions/gcs/AddGCSRecord

```
// AddGCSRecord()
// =====
// Generates and then writes a record to the current Guarded
// control stack.

AddGCSRecord(bits(64) vaddress)
    bits(64) ptr;
    AccessDescriptor accdesc = CreateAccDescGCS(PSTATE.EL, MemOp\_STORE);

    ptr = GetCurrentGCSPtr();

    // Store the record
    Mem[ptr-8, 8, accdesc] = vaddress;

    // Decrement the pointer value
    ptr = ptr - 8;

    SetCurrentGCSPtr(ptr);
    return;
```

## Library pseudocode for aarch64/functions/gcs/CheckGCSExRecord

```
// CheckGCSExRecord()
// =====
// Validates the provided values against the top entry of the
// current Guarded control stack.

CheckGCSExRecord(bits(64) elr, bits(64) spsr, bits(64) lr, GCSInstruction gcsinst_type)
    bits(64) ptr;
    AccessDescriptor accdesc = CreateAccDescGCS(PSTATE.EL, MemOp\_LOAD);
    ptr = GetCurrentGCSPtr();

    // Check the lowest doubleword is correctly formatted
    bits(64) recorded_first_dword = Mem[ptr, 8, accdesc];
    if recorded_first_dword != Zeros(60):'1001' then
        GCSDDataCheckException(gcsinst_type);

    // Check the ELR matches the recorded value
    bits(64) recorded_elr = Mem[ptr+8, 8, accdesc];
    if recorded_elr != elr then
        GCSDDataCheckException(gcsinst_type);

    // Check the SPSR matches the recorded value
    bits(64) recorded_spsr = Mem[ptr+16, 8, accdesc];
    if recorded_spsr != spsr then
        GCSDDataCheckException(gcsinst_type);

    // Check the LR matches the recorded value
    bits(64) recorded_lr = Mem[ptr+24, 8, accdesc];
    if recorded_lr != lr then
        GCSDDataCheckException(gcsinst_type);

    // Increment the pointer value
    ptr = ptr + 32;

    SetCurrentGCSPtr(ptr);
    return;
```



## Library pseudocode for aarch64/functions/gcs/CheckGCSSTREnabled

```
// CheckGCSSTREnabled()
// =====
// Trap GCSSTR or GCSSTTR instruction if trapping is enabled.

CheckGCSSTREnabled()
  case PSTATE.EL of
    when EL0
      if GCSCRE0_EL1.STREn == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
          GCSSTRTrapException(EL2);
        else
          GCSSTRTrapException(EL1);
      when EL1
        if GCSCR_EL1.STREn == '0' then
          GCSSTRTrapException(EL1);
        elsif (EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
          HFGITR_EL2.nGCSSTR_EL1 == '0') then
          GCSSTRTrapException(EL2);
      when EL2
        if GCSCR_EL2.STREn == '0' then
          GCSSTRTrapException(EL2);
      when EL3
        if GCSCR_EL3.STREn == '0' then
          GCSSTRTrapException(EL3);
  return;
```

## Library pseudocode for aarch64/functions/gcs/EXLOCKException

```
// EXLOCKException()
// =====
// Handle an EXLOCK exception condition.

EXLOCKException()
  bits(64) preferred_exception_return = ThisInstrAddr(64);
  integer vect_offset = 0x0;

  except = ExceptionSyndrome(Exception\_GCSFail);
  except.syndrome<24> = Zeros();
  except.syndrome<23:20> = '0001';
  except.syndrome<19:0> = Zeros();
  AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/functions/gcs/GCSDataCheckException

```
// GCSDataCheckException()
// =====
// Handle a Guarded Control Stack data check fault condition.

GCSDataCheckException(GCSInstruction gcsinst_type)
    bits(2) target_el;
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;
    boolean rn_unknown = FALSE;
    boolean is_ret = FALSE;

    if PSTATE.EL == EL0 then
        target_el = if (EL2Enabled() && HCR_EL2.TGE == '1') then EL2 else EL1;
    else
        target_el = PSTATE.EL;
    except = ExceptionSyndrome(Exception\_GCSFail);
    case gcsinst_type of
        when GCSInstType\_PRET
            except.syndrome<4:0> = '00000';
            is_ret = TRUE;
        when GCSInstType\_POPM
            except.syndrome<4:0> = '00001';
        when GCSInstType\_PRETAA
            except.syndrome<4:0> = '00010';
            is_ret = TRUE;
        when GCSInstType\_PRETAB
            except.syndrome<4:0> = '00011';
            is_ret = TRUE;
        when GCSInstType\_SS1
            except.syndrome<4:0> = '00100';
        when GCSInstType\_SS2
            except.syndrome<4:0> = '00101';
            rn_unknown = TRUE;
        when GCSInstType\_POPCX
            rn_unknown = TRUE;
            except.syndrome<4:0> = '01000';
        when GCSInstType\_POPX
            except.syndrome<4:0> = '01001';
    if rn_unknown == TRUE then
        except.syndrome<9:5> = bits(5) UNKNOWN;
    elsif is_ret == TRUE then
        except.syndrome<9:5> = ThisInstr()<9:5>;
    else
        except.syndrome<9:5> = ThisInstr()<4:0>;
    except.syndrome<24:10> = Zeros();
    except.vaddress = bits(64) UNKNOWN;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

### Library pseudocode for aarch64/functions/gcs/GCSEnabled

```
// GCSEnabled()
// =====
// Returns TRUE if the Guarded control stack is enabled at
// the provided Exception level.

boolean GCSEnabled(bits(2) el)
    if UsingAArch32() then
        return FALSE;

    if HaveEL(EL3) && el != EL3 && SCR_EL3.GCSEn == '0' then
        return FALSE;

    if (el IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0') &&
        (!IsHCRXEL2Enabled() || HCRX_EL2.GCSEn == '0')) then
        return FALSE;

    return GCSPCRSelected(el);
```

### Library pseudocode for aarch64/functions/gcs/GCSInstruction

```
// GCSInstruction
// =====

enumeration GCSInstruction {
    GCSInstType_PRET,    // Procedure return without Pointer authentication
    GCSInstType_POPM,    // GCSPOPM instruction
    GCSInstType_PRETAA,  // Procedure return with Pointer authentication that used key A
    GCSInstType_PRETAB,  // Procedure return with Pointer authentication that used key B
    GCSInstType_SS1,     // GCSSS1 instruction
    GCSInstType_SS2,     // GCSSS2 instruction
    GCSInstType_POPCX,   // GCSPOPCX instruction
    GCSInstType_POPX     // GCSPOPX instruction
};
```

### Library pseudocode for aarch64/functions/gcs/GCSPCREnabled

```
// GCSPCREnabled()
// =====
// Returns TRUE if the Guarded control stack is PCR enabled
// at the provided Exception level.

boolean GCSPCREnabled(bits(2) el)
    return GCSPCRSelected(el) && GCSEnabled(el);
```

### Library pseudocode for aarch64/functions/gcs/GCSPCRSelected

```
// GCSPCRSelected()
// =====
// Returns TRUE if the Guarded control stack is PCR selected
// at the provided Exception level.

boolean GCSPCRSelected(bits(2) el)
    case el of
        when EL0 return GCSCRE0_EL1.PCRSEL == '1';
        when EL1 return GCSCR_EL1.PCRSEL == '1';
        when EL2 return GCSCR_EL2.PCRSEL == '1';
        when EL3 return GCSCR_EL3.PCRSEL == '1';
    Unreachable();
    return TRUE;
```

## Library pseudocode for aarch64/functions/gcs/GCSPOPCX

```
// GCSPOPCX()
// =====
// Called to pop and compare a Guarded control stack exception return record.

GCSPOPCX()
    bits(64) spsr = SPSR\_ELx\[\];
    if !GCSEnabled(PSTATE.EL) then
        EndOfInstruction();
    CheckGCSExRecord(ELR_ELx[], spsr, X[30,64], GCSInstType\_POPCX);
    PSTATE.EXLOCK = if GetCurrentEXLOCKEN() then '1' else '0';
    return;
```

## Library pseudocode for aarch64/functions/gcs/GCSPOPM

```
// GCSPOPM()
// =====
// Called to pop a Guarded control stack procedure return record.

bits(64) GCSPOPM()
    bits(64) ptr;
    AccessDescriptor accdesc = CreateAccDescGCS(PSTATE.EL, MemOp\_LOAD);

    if !GCSEnabled(PSTATE.EL) then EndOfInstruction();
    ptr = GetCurrentGCSPtr();
    bits(64) gcs_entry = Mem[ptr, 8, accdesc];

    if gcs_entry<1:0> != '00' then
        GCSDataCheckException(GCSInstType\_POPM);

    ptr = ptr + 8;
    SetCurrentGCSPtr(ptr);
    return gcs_entry;
```

## Library pseudocode for aarch64/functions/gcs/GCSPOPX

```
// GCSPOPX()
// =====
// Called to pop a Guarded control stack exception return record.

GCSPOPX()
    if !GCSEnabled(PSTATE.EL) then EndOfInstruction();

    bits(64) ptr;
    AccessDescriptor accdesc = CreateAccDescGCS(PSTATE.EL, MemOp\_LOAD);
    ptr = GetCurrentGCSPtr();

    // Check the lowest doubleword is correctly formatted
    bits(64) recorded_first_dword = Mem[ptr, 8, accdesc];
    if recorded_first_dword != Zeros(60):'1001' then
        GCSDataCheckException(GCSInstType\_POPX);

    // Ignore these loaded values, however they might have
    // faulted which is why we load them anyway
    bits(64) recorded_elr = Mem[ptr+8, 8, accdesc];
    bits(64) recorded_spsr = Mem[ptr+16, 8, accdesc];
    bits(64) recorded_lr = Mem[ptr+24, 8, accdesc];

    // Increment the pointer value
    ptr = ptr + 32;

    SetCurrentGCSPtr(ptr);
    return;
```

## Library pseudocode for aarch64/functions/gcs/GCSPUSHM

```
// GCSPUSHM()
// =====
// Called to push a Guarded control stack procedure return record.

GCSPUSHM(bits(64) value)
    if !GCSEnabled(PSTATE.EL) then EndOfInstruction();
    AddGCSRecord(value);
    return;
```

## Library pseudocode for aarch64/functions/gcs/GCSPUSHX

```
// GCSPUSHX()
// =====
// Called to push a Guarded control stack exception return record.

GCSPUSHX()
    bits(64) spsr = SPSR_ELx[];
    if !GCSEnabled(PSTATE.EL) then
        EndOfInstruction();
    AddGCSExRecord(ELR_ELx[], spsr, X[30,64]);
    PSTATE.EXLOCK = '0';
    return;
```

## Library pseudocode for aarch64/functions/gcs/GCSReturnValueCheckEnabled

```
// GCSReturnValueCheckEnabled()
// =====
// Returns TRUE if the Guarded control stack has return value
// checking enabled at the current Exception level.

boolean GCSReturnValueCheckEnabled(bits(2) el)
    if UsingAArch32() then
        return FALSE;
    case el of
        when EL0 return GCSCRE0_EL1.RVCHKEN == '1';
        when EL1 return GCSCR_EL1.RVCHKEN == '1';
        when EL2 return GCSCR_EL2.RVCHKEN == '1';
        when EL3 return GCSCR_EL3.RVCHKEN == '1';
```

## Library pseudocode for aarch64/functions/gcs/GCSSS1

```
// GCSSS1()
// =====
// Operational pseudocode for GCSSS1 instruction.

GCSSS1(bits(64) incoming_pointer)
    bits(64) outgoing_pointer, cmpoperand, operand, data;
    if !GCSEnabled(PSTATE.EL) then EndOfInstruction();
    AccessDescriptor accdesc = CreateAccDescGCSSS1(PSTATE.EL);
    outgoing_pointer = GetCurrentGCSPointer();
    // Valid cap entry is expected
    cmpoperand = incoming_pointer[63:12]:'000000000001';
    // In-progress cap entry should be stored if the comparison is successful
    operand = outgoing_pointer[63:3]:'101';

    data = MemAtomic(incoming_pointer, cmpoperand, operand, accdesc);
    if data == cmpoperand then
        SetCurrentGCSPointer(incoming_pointer[63:3]:'000');
    else
        GCSDataCheckException(GCSInstType_SS1);
    return;
```

## Library pseudocode for aarch64/functions/gcs/GCSSS2

```
// GCSSS2()
// =====
// Operational pseudocode for GCSSS2 instruction.

bits(64) GCSSS2()
    bits(64) outgoing_pointer, incoming_pointer, outgoing_value;
    AccessDescriptor accdesc_ld = CreateAccDescGCS(PSTATE.EL, MemOp\_LOAD);
    AccessDescriptor accdesc_st = CreateAccDescGCS(PSTATE.EL, MemOp\_STORE);
    if !GCSEnabled(PSTATE.EL) then EndOfInstruction();
    incoming_pointer = GetCurrentGCSPointer();
    outgoing_value = Mem[incoming_pointer, 8, accdesc_ld];

    if outgoing_value[2:0] == '101' then //in_progress token
        outgoing_pointer[63:3] = outgoing_value[63:3] - 1;
        outgoing_pointer[2:0] = '000';
        outgoing_value = outgoing_pointer[63:12]: '000000000001';
        Mem[outgoing_pointer, 8, accdesc_st] = outgoing_value;
        SetCurrentGCSPointer(incoming_pointer + 8);
        GCSSynchronizationBarrier();
    else
        GCSDataCheckException(GCSInstType\_SS2);
    return outgoing_pointer;
```

## Library pseudocode for aarch64/functions/gcs/GCSSTRTrapException

```
// GCSSTRTrapException()
// =====
// Handle a trap on GCSSTR instruction condition.

GCSSTRTrapException(bits(2) target_el)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;

    except = ExceptionSyndrome(Exception\_GCSFail);
    except.syndrome<24> = Zeros();
    except.syndrome<23:20> = '0010';
    except.syndrome<19:15> = Zeros();
    except.syndrome<14:10> = ThisInstr()<9:5>;
    except.syndrome<9:5> = ThisInstr()<4:0>;
    except.syndrome<4:0> = Zeros();
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/functions/gcs/GCSSynchronizationBarrier

```
// GCSSynchronizationBarrier()
// =====
// Barrier instruction that synchronizes Guarded Control Stack
// accesses in relation to other load and store accesses

GCSSynchronizationBarrier();
```

## Library pseudocode for aarch64/functions/gcs/GetCurrentEXLOCKEN

```
// GetCurrentEXLOCKEN()
// =====

boolean GetCurrentEXLOCKEN()
    if Halted\(\) || Restarting\(\) then
        return FALSE;

    case PSTATE.EL of
        when EL0
            Unreachable\(\);
        when EL1
            return GCSCR_EL1.EXLOCKEN == '1';
        when EL2
            return GCSCR_EL2.EXLOCKEN == '1';
        when EL3
            return GCSCR_EL3.EXLOCKEN == '1';
```

## Library pseudocode for aarch64/functions/gcs/GetCurrentGCSPtr

```
// GetCurrentGCSPtr()
// =====
// Returns the value of the current Guarded control stack
// pointer register.

bits(64) GetCurrentGCSPtr()
    bits(64) ptr;

    case PSTATE.EL of
        when EL0
            ptr = GCSPR_EL0.PTR:'000';
        when EL1
            ptr = GCSPR_EL1.PTR:'000';
        when EL2
            ptr = GCSPR_EL2.PTR:'000';
        when EL3
            ptr = GCSPR_EL3.PTR:'000';
    return ptr;
```

## Library pseudocode for aarch64/functions/gcs/LoadCheckGCSRecord

```
// LoadCheckGCSRecord()
// =====
// Validates the provided address against the top entry of the
// current Guarded control stack.

bits(64) LoadCheckGCSRecord(bits(64) vaddress, GCSInstruction gcsinst_type)
    bits(64) ptr;
    bits(64) recorded_va;
    AccessDescriptor accdesc = CreateAccDescGCS(PSTATE.EL, MemOp\_LOAD);

    ptr = GetCurrentGCSPtr();
    recorded_va = Mem[ptr, 8, accdesc];
    if GCSReturnValueCheckEnabled(PSTATE.EL) && (recorded_va != vaddress) then
        GCSDataCheckException(gcsinst_type);

    return recorded_va;
```

## Library pseudocode for aarch64/functions/gcs/SetCurrentGCSPointer

```
// SetCurrentGCSPointer()
// =====
// Writes a value to the current Guarded control stack pointer register.

SetCurrentGCSPointer(bits(64) ptr)
  case PSTATE.EL of
    when EL0
      GCSPR_EL0.PTR = ptr<63:3>;
    when EL1
      GCSPR_EL1.PTR = ptr<63:3>;
    when EL2
      GCSPR_EL2.PTR = ptr<63:3>;
    when EL3
      GCSPR_EL3.PTR = ptr<63:3>;
  return;
```





```

// AArch64.IC()
// =====
// Perform Instruction Cache Operation.

AArch64.IC(CacheOpScope opscope)
    regval = bits(64) UNKNOWN;
    AArch64.IC(regval, opscope);

// AArch64.IC()
// =====
// Perform Instruction Cache Operation.

AArch64.IC(bits(64) regval, CacheOpScope opscope)
    CacheRecord cache;

    cache.acctype = AccessType_IC;
    cache.cachetype = CacheType_Instruction;
    cache.cacheop = CacheOp_Invalidate;
    cache.opscope = opscope;

    if opscope IN {CacheOpScope_ALLU, CacheOpScope_ALLUIS} then
        ss = SecurityStateAtEL(PSTATE.EL);
        cache.cpas = CPASAtSecurityState(ss);
        if (opscope == CacheOpScope_ALLUIS || (opscope == CacheOpScope_ALLU && PSTATE.EL == EL1
            && EL2Enabled() && HCR_EL2.FB == '1')) then
            cache.shareability = Shareability_ISH;
        else
            cache.shareability = Shareability_NSH;
        cache.regval = regval;
        CACHE_OP(cache);
    else
        assert opscope == CacheOpScope_PoU;

        if EL2Enabled() && !IsInHost() then
            if PSTATE.EL IN {EL0, EL1} then
                cache.is_vmid_valid = TRUE;
                cache.vmid = VMID[];
            else
                cache.is_vmid_valid = FALSE;
        else
            cache.is_vmid_valid = FALSE;

        if PSTATE.EL == EL0 then
            cache.is_asid_valid = TRUE;
            cache.asid = ASID[];
        else
            cache.is_asid_valid = FALSE;

        bits(64) vaddress = regval;
        boolean need_translate = ICInstNeedsTranslation(opscope);

        cache.vaddress = regval;
        cache.shareability = Shareability_NSH;
        cache.translated = need_translate;

        if !need_translate then
            cache.paddress = FullAddress UNKNOWN;
            CACHE_OP(cache);
            return;

        AccessDescriptor accdesc = CreateAccDescIC(cache);
        boolean aligned = TRUE;
        integer size = 0;
        AddressDescriptor memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);

        if IsFault(memaddrdesc) then
            AArch64.Abort(regval, memaddrdesc.fault);

        cache.cpas = CPASAtPAS(memaddrdesc.paddress.paspace);
        cache.paddress = memaddrdesc.paddress;

```

```

    CACHE\_OP(cache);
return;

```

### Library pseudocode for aarch64/functions/immediateop/ImmediateOp

```

// ImmediateOp
// =====
// Vector logical immediate instruction types.

enumeration ImmediateOp {ImmediateOp_MOVI, ImmediateOp_MVNI,
    ImmediateOp_ORR, ImmediateOp_BIC};

```

### Library pseudocode for aarch64/functions/logicalop/LogicalOp

```

// LogicalOp
// =====
// Logical instruction types.

enumeration LogicalOp {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};

```

### Library pseudocode for aarch64/functions/mec/AArch64.S1AMECFault

```

// AArch64.S1AMECFault()
// =====
// Returns TRUE if a Translation fault should occur for Realm EL2 and Realm EL2&0
// stage 1 translated addresses to Realm PA space.

boolean AArch64.S1AMECFault(S1TTWParams walkparams, PASpace paspace, Regime regime,
    bits(N) descriptor)
    assert N IN {64,128};
    bit descriptor_amec = if walkparams.d128 == '1' then descriptor<103> else descriptor<63>;

    return (walkparams.<emec,amec> == '10' &&
        regime IN {Regime\_EL2, Regime\_EL20} &&
        paspace == PAS Realm &&
        descriptor_amec == '1');

```

### Library pseudocode for aarch64/functions/mec/AArch64.S1DisabledOutputMECID

```

// AArch64.S1DisabledOutputMECID()
// =====
// Returns the output MECID when stage 1 address translation is disabled.

bits(16) AArch64.S1DisabledOutputMECID(S1TTWParams walkparams, Regime regime, PASpace paspace)
    if walkparams.emec == '0' then
        return DEFAULT\_MECID;

    if !(regime IN {Regime\_EL2, Regime\_EL20, Regime\_EL10}) then
        return DEFAULT\_MECID;

    if paspace != PAS Realm then
        return DEFAULT\_MECID;

    if regime == Regime\_EL10 then
        return VMECID_P_EL2.MECID;
    else
        return MECID_P0_EL2.MECID;

```

## Library pseudocode for aarch64/functions/mec/AArch64.S1OutputMECID

```
// AArch64.S1OutputMECID()
// =====
// Returns the output MECID when stage 1 address translation is enabled.

bits(16) AArch64.S1OutputMECID(S1TTWParams walkparams, Regime regime, VARange varange,
                                PASpace paspace, bits(N) descriptor)

    assert N IN {64,128};

    if walkparams.emec == '0' then
        return DEFAULT MECID;

    if paspace != PAS Realm then
        return DEFAULT MECID;

    bit descriptor_amec = if walkparams.d128 == '1' then descriptor<103> else descriptor<63>;
    case regime of
        when Regime EL3
            return MECID_RL_A_EL3.MECID;
        when Regime EL2
            if descriptor_amec == '0' then
                return MECID_P0_EL2.MECID;
            else
                return MECID_A0_EL2.MECID;
        when Regime EL20
            if varange == VARange LOWER then
                if descriptor_amec == '0' then
                    return MECID_P0_EL2.MECID;
                else
                    return MECID_A0_EL2.MECID;
            else
                if descriptor_amec == '0' then
                    return MECID_P1_EL2.MECID;
                else
                    return MECID_A1_EL2.MECID;
        when Regime EL10
            return VMECID_P_EL2.MECID;
```

## Library pseudocode for aarch64/functions/mec/AArch64.S2OutputMECID

```
// AArch64.S2OutputMECID()
// =====
// Returns the output MECID for stage 2 address translation.

bits(16) AArch64.S2OutputMECID(S2TTWParams walkparams, PASpace paspace, bits(N) descriptor)

    assert N IN {64,128};

    if walkparams.emec == '0' then
        return DEFAULT MECID;

    if paspace != PAS Realm then
        return DEFAULT MECID;

    bit descriptor_amec = if walkparams.d128 == '1' then descriptor<103> else descriptor<63>;
    if descriptor_amec == '0' then
        return VMECID_P_EL2.MECID;
    else
        return VMECID_A_EL2.MECID;
```

## Library pseudocode for aarch64/functions/mec/AArch64.TTWalkMECID

```
// AArch64.TTWalkMECID()
// =====
// Returns the associated MECID for the translation table walk of the given
// translation regime and Security state.

bits(16) AArch64.TTWalkMECID(bit emec, Regime regime, SecurityState ss)
    if emec == '0' then
        return DEFAULT\_MECID;

    if ss != SS\_Realm then
        return DEFAULT\_MECID;

    case regime of
        when Regime\_EL2
            return MECID_P0_EL2.MECID;
        when Regime\_EL20
            if TCR_EL2.A1 == '0' then
                return MECID_P1_EL2.MECID;
            else
                return MECID_P0_EL2.MECID;
        // This applies to stage 1 and stage 2 translation table walks for
        // Realm EL1&0, but the stage 2 translation for a stage 1 walk
        // might later override the MECID according to AMEC configuration.
        when Regime\_EL10
            return VMESID_P_EL2.MECID;
        otherwise
            Unreachable();
```

## Library pseudocode for aarch64/functions/mec/DEFAULT\_MECID

```
constant bits(16) DEFAULT_MECID = Zeros(16);
```

## Library pseudocode for aarch64/functions/memory/AArch64.AccessIsTagChecked

```
// AArch64.AccessIsTagChecked()
// =====
// TRUE if a given access is tag-checked, FALSE otherwise.

boolean AArch64.AccessIsTagChecked(bits(64) vaddr, AccessDescriptor accdesc)
    assert accdesc.tagchecked;

    if UsingAArch32() then
        return FALSE;

    boolean is_instr = FALSE;
    if (EffectiveMTX(vaddr, is_instr, PSTATE.EL) == '0' &&
        EffectiveTBI(vaddr, is_instr, PSTATE.EL) == '0') then
        return FALSE;

    if (EffectiveTCMA(vaddr, PSTATE.EL) == '1' &&
        (vaddr<59:55> == '00000' || vaddr<59:55> == '11111')) then
        return FALSE;

    if !AArch64.AllocationTagAccessIsEnabled(accdesc.el) then
        return FALSE;

    if PSTATE.TCO=='1' then
        return FALSE;

    if HaveMTEStoreOnlyExt() && !accdesc.write && StoreOnlyTagCheckingEnabled(accdesc.el) then
        return FALSE;

    return TRUE;
```

### Library pseudocode for aarch64/functions/memory/AArch64.AddressWithAllocationTag

```
// AArch64.AddressWithAllocationTag()
// =====
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit
// virtual address and an Allocation Tag.
// If the extension is disabled, treats the Allocation Tag as '0000'.

bits(64) AArch64.AddressWithAllocationTag(bits(64) address, bits(4) allocation_tag)
    bits(64) result = address;
    bits(4) tag;
    if AArch64.AllocationTagAccessIsEnabled(PSTATE.EL) then
        tag = allocation_tag;
    else
        tag = '0000';
    result<59:56> = tag;
    return result;
```

### Library pseudocode for aarch64/functions/memory/AArch64.AllocationTagCheck

```
// AArch64.AllocationTagCheck()
// =====
// Performs an Allocation Tag Check operation for a memory access and
// returns whether the check passed.

boolean AArch64.AllocationTagCheck(AddressDescriptor memaddrdesc, AccessDescriptor accdesc,
                                   bits(4) ptag)
    if memaddrdesc.memattrs.tags == MemTag\_AllocationTagged then
        (memstatus, readtag) = PhysMemTagRead(memaddrdesc, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);

        return ptag == readtag;
    else
        return TRUE;
```

### Library pseudocode for aarch64/functions/memory/AArch64.AllocationTagFromAddress

```
// AArch64.AllocationTagFromAddress()
// =====
// Generate an Allocation Tag from a 64-bit value containing a Logical Address Tag.

bits(4) AArch64.AllocationTagFromAddress(bits(64) tagged_address)
    return tagged_address<59:56>;
```

### Library pseudocode for aarch64/functions/memory/AArch64.CanonicalTagCheck

```
// AArch64.CanonicalTagCheck()
// =====
// Performs a Canonical Tag Check operation for a memory access and
// returns whether the check passed.

boolean AArch64.CanonicalTagCheck(AddressDescriptor memaddrdesc, bits(4) ptag)
    expected_tag = if memaddrdesc.vaddress<55> == '0' then '0000' else '1111';
    return ptag == expected_tag;
```

## Library pseudocode for aarch64/functions/memory/AArch64.CheckTag

```
// AArch64.CheckTag()
// =====
// Performs a Tag Check operation for a memory access and returns
// whether the check passed

boolean AArch64.CheckTag(AddressDescriptor memaddrdesc, AccessDescriptor accdesc, bits(4) ptag)
    if memaddrdesc.memattrs.tags == MemTag\_AllocationTagged then
        return AArch64.AllocationTagCheck(memaddrdesc, accdesc, ptag);
    elsif memaddrdesc.memattrs.tags == MemTag\_CanonicalyTagged then
        return AArch64.CanonicalTagCheck(memaddrdesc, ptag);
    else
        return TRUE;
```

## Library pseudocode for aarch64/functions/memory/AArch64.IsUnprivAccessPriv

```
// AArch64.IsUnprivAccessPriv()
// =====
// Returns TRUE if an unprivileged access is privileged, and FALSE otherwise.

boolean AArch64.IsUnprivAccessPriv()
    boolean privileged;

    case PSTATE.EL of
        when EL0 privileged = FALSE;
        when EL1 privileged = EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11';
        when EL2 privileged = !(HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11');
        when EL3 privileged = TRUE;

    if HaveUAOExt() && PSTATE.UAO == '1' then
        privileged = PSTATE.EL != EL0;

    return privileged;
```





```

// AArch64.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccessDescriptor accdesc_in,
                                     boolean aligned]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    AccessDescriptor accdesc = accdesc_in;
    if HaveLSE2Ext() then
        assert AllInAlignedQuantity(address, size, 16);
    else
        assert IsAligned(address, size);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if HaveMTE2Ext() && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    AddressDescriptor memaddrdesc;
    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Memory array access
    if HaveTME() then
        if accdesc.transactional && !MemHasTransactionalAccess(memaddrdesc.memattrs) then
            FailTransaction(TMFailure\_IMP, FALSE);

    if HaveMTE2Ext() && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            AArch64.TagCheckFault(address, accdesc);

    if SPESampleInFlight then
        boolean is_load = TRUE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);

    boolean atomic;
    if (memaddrdesc.memattrs.memtype == MemType\_Normal &&
        memaddrdesc.memattrs.inner.attrs == MemAttr\_WB &&
        memaddrdesc.memattrs.outer.attrs == MemAttr\_WB) then
        atomic = TRUE;
    elseif (accdesc.exclusive || accdesc.atomicop ||
            accdesc.acqsc || accdesc.acqpc || accdesc.relsc) then
        if !aligned && !ConstrainUnpredictableBool(Unpredictable\_MISALIGNEDATOMIC) then
            AArch64.Abort(address, AlignmentFault(accdesc));
        else
            atomic = TRUE;
    elseif aligned then
        atomic = !accdesc.ispair;
    else
        // Misaligned accesses within 16 byte aligned memory but
        // not Normal Cacheable Writeback are Atomic
        atomic = boolean IMPLEMENTATION_DEFINED "FEAT_LSE2: access is atomic";

    PhysMemRetStatus memstatus;
    if atomic then
        (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
    elseif aligned && accdesc.ispair then
        assert size IN {8, 16};
        constant halfsize = size DIV 2;
        bits(halfsize * 8) lowhalf, highhalf;
        (memstatus, lowhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);

```

```

    memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
    (memstatus, highhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);

    value = highhalf:lowhalf;
else
    for i = 0 to size-1
        (memstatus, value<8*i+7:8*i>) = PhysMemRead(memaddrdesc, 1, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    return value;

// AArch64.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccessDescriptor accdesc_in,
    boolean aligned] = bits(size*8) value
assert size IN {1, 2, 4, 8, 16};
AccessDescriptor accdesc = accdesc_in;
if HaveLSE2Ext() then
    assert AllInAlignedQuantity(address, size, 16);
else
    assert IsAligned(address, size);

// If the instruction encoding permits tag checking, confer with system register configuration
// which may override this.
if HaveMTE2Ext() && accdesc.tagchecked then
    accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

AddressDescriptor memaddrdesc;
memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability\_NSH then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

if HaveTME() then
    if accdesc.transactional && !MemHasTransactionalAccess(memaddrdesc.memattrs) then
        FailTransaction(TMFailure\_IMP, FALSE);

if HaveMTE2Ext() && accdesc.tagchecked then
    bits(4) ptag = AArch64.PhysicalTag(address);
    if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
        AArch64.TagCheckFault(address, accdesc);

if SPESampleInFlight then
    boolean is_load = FALSE;
    SPESampleLoadStore(is_load, accdesc, memaddrdesc);

PhysMemRetStatus memstatus;
boolean atomic;
if (memaddrdesc.memattrs.memtype == MemType\_Normal &&
    memaddrdesc.memattrs.inner.attrs == MemAttr\_WB &&
    memaddrdesc.memattrs.outer.attrs == MemAttr\_WB) then
    atomic = TRUE;
elseif (accdesc.exclusive || accdesc.atomicop ||
    accdesc.acqsc || accdesc.acqpc || accdesc.relsc) then
    if !aligned && !ConstrainUnpredictableBool(Unpredictable\_MISALIGNEDATOMIC) then
        AArch64.Abort(address, AlignmentFault(accdesc));
    else
        atomic = TRUE;
elseif aligned then
    atomic = !accdesc.ispair;

```

```

else
    // Misaligned accesses within 16 byte aligned memory but
    // not Normal Cacheable Writeback are Atomic
    atomic = boolean IMPLEMENTATION_DEFINED "FEAT_LSE2: access is atomic";

if atomic then
    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
elseif aligned && accdesc.ispair then
    assert size IN {8, 16};
    constant halfsize = size DIV 2;
    bits(halfsize*8) lowhalf, highhalf;
    <highhalf, lowhalf> = value;

    memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, lowhalf);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
    memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
    memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, highhalf);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
else
    for i = 0 to size-1
        memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, value<8*i+7:8*i>);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
return;

```

## Library pseudocode for aarch64/functions/memory/AArch64.MemTag

```
// AArch64.MemTag[] - non-assignment (read) form
// =====
// Load an Allocation Tag from memory.

bits(4) AArch64.MemTag[bits(64) address, AccessDescriptor accdesc_in]
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

    AddressDescriptor memaddrdesc;
    AccessDescriptor accdesc = accdesc_in;

    boolean aligned = TRUE;

    if HaveMTE2Ext() then
        accdesc.tagaccess = AArch64.AllocationTagAccessIsEnabled(accdesc.el);

    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, TAG\_GRANULE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Return the granule tag if tagging is enabled...
    if accdesc.tagaccess && memaddrdesc.memattrs.tags == MemTag\_AllocationTagged then
        (memstatus, tag) = PhysMemTagRead(memaddrdesc, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
        return tag;
    elseif (HaveMTECanonicalTagCheckingExt() &&
        accdesc.tagaccess &&
        memaddrdesc.memattrs.tags == MemTag\_CanonicalTagged) then
        return if address<55> == '0' then '0000' else '1111';
    else
        // ...otherwise read tag as zero.
        return '0000';

// AArch64.MemTag[] - assignment (write) form
// =====
// Store an Allocation Tag to memory.

AArch64.MemTag[bits(64) address, AccessDescriptor accdesc_in] = bits(4) value
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

    AddressDescriptor memaddrdesc;
    AccessDescriptor accdesc = accdesc_in;

    boolean aligned = IsAligned(address, TAG\_GRANULE);

    // Stores of allocation tags must be aligned
    if !aligned then
        AArch64.Abort(address, AlignmentFault(accdesc));

    if HaveMTE2Ext() then
        accdesc.tagaccess = AArch64.AllocationTagAccessIsEnabled(accdesc.el);

    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, TAG\_GRANULE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Memory array access
    if accdesc.tagaccess && memaddrdesc.memattrs.tags == MemTag\_AllocationTagged then
        memstatus = PhysMemTagWrite(memaddrdesc, accdesc, value);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
```

### Library pseudocode for aarch64/functions/memory/AArch64.PhysicalTag

```
// AArch64.PhysicalTag()
// =====
// Generate a Physical Tag from a Logical Tag in an address

bits(4) AArch64.PhysicalTag(bits(64) vaddr)
    return vaddr<59:56>;
```

### Library pseudocode for aarch64/functions/memory/AArch64.UnalignedAccessFaults

```
// AArch64.UnalignedAccessFaults()
// =====
// Determine whether the unaligned access generates an Alignment fault

boolean AArch64.UnalignedAccessFaults(AccessDescriptor accdesc, bits(64) address, integer size)
    if AlignmentEnforced() then
        return TRUE;
    elsif accdesc.acctype == AccessType\_GCS then
        return TRUE;
    elsif accdesc.rcw then
        return TRUE;
    elsif accdesc.ls64 then
        return TRUE;
    elsif accdesc.exclusive || accdesc.atomicop then
        return !HaveLSE2Ext() || !AllInAlignedQuantity(address, size, 16);
    elsif accdesc.acqsc || accdesc.acqpc || accdesc.relsc then
        return (!HaveLSE2Ext() ||
            (SCTLR\_ELx[][.nAA == '0' && !AllInAlignedQuantity(address, size, 16))));
    else
        return FALSE;
```

### Library pseudocode for aarch64/functions/memory/AddressSupportsLS64

```
// AddressSupportsLS64()
// =====
// Returns TRUE if the 64-byte block following the given address supports the
// LD64B and ST64B instructions, and FALSE otherwise.

boolean AddressSupportsLS64(bits(56) paddress);
```

### Library pseudocode for aarch64/functions/memory/AllInAlignedQuantity

```
// AllInAlignedQuantity()
// =====
// Returns TRUE if all accessed bytes are within one aligned quantity, FALSE otherwise.

boolean AllInAlignedQuantity(bits(64) address, integer size, integer alignment)
    assert(size <= alignment);
    return Align((address+size)-1, alignment) == Align(address, alignment);
```

## Library pseudocode for aarch64/functions/memory/CheckSPAlignment

```
// CheckSPAlignment()
// =====
// Check correct stack pointer alignment for AArch64 state.

CheckSPAlignment()
    bits(64) sp = SP[];
    boolean stack_align_check;
    if PSTATE.EL == EL0 then
        stack_align_check = (SCTLR\_ELx[][.SA0 != '0']);
    else
        stack_align_check = (SCTLR\_ELx[][.SA != '0']);

    if stack_align_check && sp != Align(sp, 16) then
        AArch64.SPAlignmentFault();

    return;
```



```

// Mem[] - non-assignment (read) form
// =====
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

bits(size*8) Mem[bits(64) address, integer size, AccessDescriptor accdesc_in]
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    bits(size * 8) value;
    bits(halfsize * 8) lowhalf, highhalf;
    AccessDescriptor accdesc = accdesc_in;

    // Check alignment on size of element accessed, not overall access size
    integer alignment = if accdesc.ispair then halfsize else size;
    boolean aligned = IsAligned(address, alignment);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    if accdesc.acctype == AccessType\_ASIMD && size == 16 && IsAligned(address, 8) then
        // If 128-bit SIMD&FP ordered access are treated as a pair of
        // 64-bit single-copy atomic accesses, then these single copy atomic
        // access can be observed in any order.
        lowhalf = AArch64.MemSingle[address, halfsize, accdesc, aligned];
        highhalf = AArch64.MemSingle[address+halfsize, halfsize, accdesc, aligned];
        value = highhalf:lowhalf;
    elseif HaveLSE2Ext() && AllInAlignedQuantity(address, size, 16) then
        value = AArch64.MemSingle[address, size, accdesc, aligned];
    elseif accdesc.ispair && aligned then
        accdesc.ispair = FALSE;
        if HaveLRCPC3Ext() && accdesc.highestaddressfirst then
            highhalf = AArch64.MemSingle[address+halfsize, halfsize, accdesc, aligned];
            lowhalf = AArch64.MemSingle[address, halfsize, accdesc, aligned];
        else
            lowhalf = AArch64.MemSingle[address, halfsize, accdesc, aligned];
            highhalf = AArch64.MemSingle[address+halfsize, halfsize, accdesc, aligned];
        value = highhalf:lowhalf;
    elseif aligned then
        value = AArch64.MemSingle[address, size, accdesc, aligned];
    else
        assert size > 1;
        if HaveLRCPC3Ext() && accdesc.ispair && accdesc.highestaddressfirst then
            // Performing memory accesses from one load or store instruction to Device memory that
            // crosses a boundary corresponding to the smallest translation granule size of the
            // implementation causes CONSTRAINED UNPREDICTABLE behavior.

            for i = 0 to halfsize-1
                // Individual byte access can be observed in any order
                highhalf<8*i+7:8*i> = AArch64.MemSingle[address+halfsize +i, 1, accdesc, aligned];
            for i = 0 to halfsize-1
                // Individual byte access can be observed in any order
                lowhalf<8*i+7:8*i> = AArch64.MemSingle[address + i, 1, accdesc, aligned];

            value = highhalf:lowhalf;

        else
            value<7:0> = AArch64.MemSingle[address, 1, accdesc, aligned];

            // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device
            // memory access will generate an Alignment Fault, as to get this far means the first
            // byte did not, so we must be changing to a new translation page.
            c = ConstrainUnpredictable(Unpredictable\_DEVPAGE2);
            assert c IN {Constraint\_FAULT, Constraint\_NONE};
            if c == Constraint\_NONE then aligned = TRUE;

            for i = 1 to size-1
                value<8*i+7:8*i> = AArch64.MemSingle[address+i, 1, accdesc, aligned];

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);

```



```

    return value;

// Mem[] - assignment (write) form
// =====
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem[bits(64) address, integer size, AccessDescriptor accdesc_in] = bits(size*8) value_in
    constant halfsize = size DIV 2;
    bits(size*8) value = value_in;
    bits(halfsize*8) lowhalf, highhalf;
    AccessDescriptor accdesc = accdesc_in;

    // Check alignment on size of element accessed, not overall access size
    integer alignment = if accdesc.ispair then halfsize else size;
    boolean aligned = IsAligned(address, alignment);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);

    if accdesc.acctype == AccessType\_ASIMD && size == 16 && IsAligned(address, 8) then
        // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        <highhalf, lowhalf> = value;
        AArch64.MemSingle[address, halfsize, accdesc, aligned] = lowhalf;
        AArch64.MemSingle[address+halfsize, halfsize, accdesc, aligned] = highhalf;
    elseif HaveLSE2Ext() && AllInAlignedQuantity(address, size, 16) then
        AArch64.MemSingle[address, size, accdesc, aligned] = value;
    elseif accdesc.ispair && aligned then
        accdesc.ispair = FALSE;
        <highhalf, lowhalf> = value;
        if HaveLRCPC3Ext() && accdesc.highestaddressfirst then
            AArch64.MemSingle[address+halfsize, halfsize, accdesc, aligned] = highhalf;
            AArch64.MemSingle[address, halfsize, accdesc, aligned] = lowhalf;
        else
            AArch64.MemSingle[address, halfsize, accdesc, aligned] = lowhalf;
            AArch64.MemSingle[address+halfsize, halfsize, accdesc, aligned] = highhalf;
    elseif aligned then
        AArch64.MemSingle[address, size, accdesc, aligned] = value;
    else
        assert size > 1;
        if HaveLRCPC3Ext() && accdesc.ispair && accdesc.highestaddressfirst then
            // Performing memory accesses from one load or store instruction to Device memory that
            // crosses a boundary corresponding to the smallest translation granule size of the
            // implementation causes CONSTRAINED UNPREDICTABLE behavior.
            <highhalf, lowhalf> = value;
            for i = 0 to halfsize-1
                // Individual byte access can be observed in any order
                AArch64.MemSingle[address+halfsize+i, 1, accdesc, aligned] = highhalf<8*i+7:8*i>;
            for i = 0 to halfsize-1
                // Individual byte access can be observed in any order, but implies observability
                // of highhalf
                AArch64.MemSingle[address+i, 1, accdesc, aligned] = lowhalf<8*i+7:8*i>;
        else
            AArch64.MemSingle[address, 1, accdesc, aligned] = value<7:0>;

            // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device
            // memory access will generate an Alignment Fault, as to get this far means the first
            // byte did not, so we must be changing to a new translation page.

        c = ConstrainUnpredictable(Unpredictable\_DEVPAGE2);
        assert c IN {Constraint\_FAULT, Constraint\_NONE};
        if c == Constraint\_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch64.MemSingle[address+i, 1, accdesc, aligned] = value<8*i+7:8*i>;

return;

```



```

// MemAtomic()
// =====
// Performs load and store memory operations for a given virtual address.

bits(size) MemAtomic(bits(64) address, bits(size) cmpoperand, bits(size) operand,
    AccessDescriptor accdesc_in)
    assert accdesc_in.atomicop;

    constant integer bytes = size DIV 8;
    assert bytes IN {1, 2, 4, 8, 16};

    bits(size) newvalue;
    bits(size) oldvalue;
    AccessDescriptor accdesc = accdesc_in;
    boolean aligned = IsAligned(address, bytes);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if HaveMTE2Ext() && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, bytes) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    // MMU or MPU lookup
    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        ClearExclusiveByAddress(memaddrdesc.address, ProcessorID(), size);

    // For Store-only Tag checking, the tag check is performed on the store.
    if (HaveMTE2Ext() && accdesc.tagchecked &&
        (!HaveMTEStoreOnlyExt() || !StoreOnlyTagCheckingEnabled(accdesc.el))) then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            accdesc.write = FALSE; // Tag Check Fault on a read
            AArch64.TagCheckFault(address, accdesc);

    // All observers in the shareability domain observe the following load and store atomically.
    PhysMemRetStatus memstatus;
    (memstatus, oldvalue) = PhysMemRead(memaddrdesc, bytes, accdesc);

    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, bytes, accdesc);
    if BigEndian(accdesc.acctype) then
        oldvalue = BigEndianReverse(oldvalue);

    boolean cmpfail = FALSE;
    case accdesc.modop of
        when MemAtomicOp\_ADD      newvalue = oldvalue + operand;
        when MemAtomicOp\_BIC      newvalue = oldvalue AND NOT(operand);
        when MemAtomicOp\_EOR      newvalue = oldvalue EOR operand;
        when MemAtomicOp\_ORR      newvalue = oldvalue OR operand;
        when MemAtomicOp\_SMAX     newvalue = Max(SInt(oldvalue), SInt(operand))<size-1:0>;
        when MemAtomicOp\_SMIN     newvalue = Min(SInt(oldvalue), SInt(operand))<size-1:0>;
        when MemAtomicOp\_UMAX     newvalue = Max(UInt(oldvalue), UInt(operand))<size-1:0>;
        when MemAtomicOp\_UMIN     newvalue = Min(UInt(oldvalue), UInt(operand))<size-1:0>;
        when MemAtomicOp\_SWP      newvalue = operand;
        when MemAtomicOp\_CAS      newvalue = operand; cmpfail = cmpoperand != oldvalue;
        when MemAtomicOp\_GCSSS1   newvalue = operand; cmpfail = cmpoperand != oldvalue;

    if HaveMTEStoreOnlyExt() && StoreOnlyTagCheckingEnabled(accdesc.el) then
        // If the compare on a CAS fails, then it is CONSTRAINED UNPREDICTABLE whether the
        // Tag check is performed.
        if accdesc.tagchecked && cmpfail then

```

```

    accdesc.tagchecked = ConstrainUnpredictableBool(Unpredictable\_STOREONLYTAGCHECKEDCAS);

    if HaveMTE2Ext() && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            AArch64.TagCheckFault(address, accdesc);

    if !cmpfail then
        if BigEndian(accdesc.acctype) then
            newvalue = BigEndianReverse(newvalue);
        memstatus = PhysMemWrite(memaddrdesc, bytes, accdesc, newvalue);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);

    if SPESampleInFlight then
        boolean is_load = FALSE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);

    // Load operations return the old (pre-operation) value
    return oldvalue;

```



```

// MemAtomicRCW()
// =====
// Perform a single-copy-atomic access with Read-Check-Write operation

(bits(4), bits(size)) MemAtomicRCW(bits(64) address, bits(size) cmpoperand, bits(size) operand,
                                   AccessDescriptor accdesc_in)

    assert accdesc_in.atomicop;
    assert accdesc_in.rcw;

    constant integer bytes = size DIV 8;
    assert bytes IN {8, 16};

    bits(4) nzcw;
    bits(size) oldvalue;
    bits(size) newvalue;
    AccessDescriptor accdesc = accdesc_in;
    boolean aligned = IsAligned(address, bytes);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if HaveMTE2Ext() && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, bytes) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    // MMU or MPU lookup
    AccessDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // For Store-only Tag checking, the tag check is performed on the store.
    if (HaveMTE2Ext() && accdesc.tagchecked &&
        (!HaveMTEStoreOnlyExt() || !StoreOnlyTagCheckingEnabled(accdesc.el))) then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            accdesc.write = FALSE; // Tag Check Fault on a read
            AArch64.TagCheckFault(address, accdesc);

    // All observers in the shareability domain observe the following load and store atomically.
    PhysMemRetStatus memstatus;
    (memstatus, oldvalue) = PhysMemRead(memaddrdesc, bytes, accdesc);

    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, bytes, accdesc);

    if BigEndian(accdesc.acctype) then
        oldvalue = BigEndianReverse(oldvalue);

    boolean cmpfail = FALSE;
    case accdesc.modop of
        when MemAtomicOp\_BIC newvalue = oldvalue AND NOT(operand);
        when MemAtomicOp\_ORR newvalue = oldvalue OR operand;
        when MemAtomicOp\_SWP newvalue = operand;
        when MemAtomicOp\_CAS newvalue = operand; cmpfail = oldvalue != cmpoperand;

    if cmpfail then
        nzcw = '1010'; // N = 1 indicates compare failure
    else
        nzcw = RCWCheck(oldvalue, newvalue, accdesc.rcws);

    if HaveMTEStoreOnlyExt() && StoreOnlyTagCheckingEnabled(accdesc.el) then
        // If the compare on a CAS fails, then it is CONSTRAINED UNPREDICTABLE whether the
        // Tag check is performed.

```

```

if accdesc.tagchecked && cmpfail then
    accdesc.tagchecked = ConstrainUnpredictableBool(Unpredictable\_STOREONLYTAGCHECKEDCAS);

if HaveMTE2Ext() && accdesc.tagchecked then
    bits(4) ptag = AArch64.PhysicalTag(address);
    if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
        AArch64.TagCheckFault(address, accdesc);

if nzcvc == '0010' then
    if BigEndian(accdesc.acctype) then
        newvalue = BigEndianReverse(newvalue);

    memstatus = PhysMemWrite(memaddrdesc, bytes, accdesc, newvalue);

    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);

return (nzcvc, oldvalue);

```

## Library pseudocode for aarch64/functions/memory/MemLoad64B

```
// MemLoad64B()
// =====
// Performs an atomic 64-byte read from a given virtual address.

bits(512) MemLoad64B(bits(64) address, AccessDescriptor accdesc_in)
    bits(512) data;
    constant integer size = 64;
    AccessDescriptor accdesc = accdesc_in;
    boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if HaveMTE2Ext() && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    if HaveMTE2Ext() && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            AArch64.TagCheckFault(address, accdesc);

    if !AddressSupportsLS64(memaddrdesc.paddress.address) then
        c = ConstrainUnpredictable(Unpredictable\_LS64UNSUPPORTED);
        assert c IN {Constraint\_LIMITED\_ATOMICITY, Constraint\_FAULT};

        if c == Constraint\_FAULT then
            // Generate a stage 1 Data Abort reported using the DFSC code of 110101.
            AArch64.Abort(address, ExclusiveFault(accdesc));
        else
            // Accesses are not single-copy atomic above the byte level.
            for i = 0 to size-1
                PhysMemRetStatus memstatus;
                (memstatus, data<8*i+7:8*i>) = PhysMemRead(memaddrdesc, 1, accdesc);
                if IsFault(memstatus) then
                    HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);

                memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    else
        PhysMemRetStatus memstatus;
        (memstatus, data) = PhysMemRead(memaddrdesc, size, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);

    return data;
```



## Library pseudocode for aarch64/functions/memory/MemStore64B

```
// MemStore64B()
// =====
// Performs an atomic 64-byte store to a given virtual address. Function does
// not return the status of the store.

MemStore64B(bits(64) address, bits(512) value, AccessDescriptor accdesc_in)
    constant integer size = 64;
    AccessDescriptor accdesc = accdesc_in;
    boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if HaveMTE2Ext() && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);

    if HaveMTE2Ext() && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            AArch64.TagCheckFault(address, accdesc);

    PhysMemRetStatus memstatus;
    if !AddressSupportsLS64(memaddrdesc.paddress.address) then
        c = ConstrainUnpredictable(Unpredictable\_LS64UNSUPPORTED);
        assert c IN {Constraint\_LIMITED\_ATOMICITY, Constraint\_FAULT};

        if c == Constraint\_FAULT then
            // Generate a Data Abort reported using the DFSC code of 110101.
            AArch64.Abort(address, ExclusiveFault(accdesc));
        else
            // Accesses are not single-copy atomic above the byte level.
            for i = 0 to size-1
                memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, value<8*i+7:8*i>);
                if IsFault(memstatus) then
                    HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);

                memaddrdesc.paddress.address = memaddrdesc.paddress.address+1;
    else
        memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);

    return;
```

## Library pseudocode for aarch64/functions/memory/MemStore64BWithRet

```
// MemStore64BWithRet()
// =====
// Performs an atomic 64-byte store to a given virtual address returning
// the status value of the operation.

bits(64) MemStore64BWithRet(bits(64) address, bits(512) value, AccessDescriptor accdesc_in)
    constant integer size = 64;
    AccessDescriptor accdesc = accdesc_in;
    boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if HaveMTE2Ext() && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);
        return ZeroExtend('1', 64);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);

    if HaveMTE2Ext() && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            AArch64.TagCheckFault(address, accdesc);
            return ZeroExtend('1', 64);

    PhysMemRetStatus memstatus;
    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);

    return memstatus.store64bstatus;
```

## Library pseudocode for aarch64/functions/memory/MemStore64BWithRetStatus

```
// MemStore64BWithRetStatus()
// =====
// Generates the return status of memory write with ST64BV or ST64BV0
// instructions. The status indicates if the operation succeeded, failed,
// or was not supported at this memory location.

bits(64) MemStore64BWithRetStatus();
```

## Library pseudocode for aarch64/functions/memory/NVMem

```
// NVMem[] - non-assignment form
// =====
// This function is the load memory access for the transformed System register read access
// when Enhanced Nested Virtualization is enabled with HCR_EL2.NV2 = 1.
// The address for the load memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
// * VNCR_EL2.BADDR holds the base address of the memory location, and
// * Offset is the unique offset value defined architecturally for each System register that
// supports transformation of register access to memory access.

bits(64) NVMem[integer offset]
    assert offset > 0;
    constant integer size = 64;
    return NVMem[offset, size];

bits(N) NVMem[integer offset, integer N]
    assert offset > 0;
    assert N IN {64,128};
    bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
    AccessDescriptor accdesc = CreateAccDescNV2 (MemOp_LOAD);
    return Mem[address, N DIV 8, accdesc];

// NVMem[] - assignment form
// =====
// This function is the store memory access for the transformed System register write access
// when Enhanced Nested Virtualization is enabled with HCR_EL2.NV2 = 1.
// The address for the store memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
// * VNCR_EL2.BADDR holds the base address of the memory location, and
// * Offset is the unique offset value defined architecturally for each System register that
// supports transformation of register access to memory access.

NVMem[integer offset] = bits(64) value
    assert offset > 0;
    constant integer size = 64;
    NVMem[offset, size] = value;
    return;

NVMem[integer offset, integer N] = bits(N) value
    assert offset > 0;
    assert N IN {64,128};
    bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
    AccessDescriptor accdesc = CreateAccDescNV2 (MemOp_STORE);
    Mem[address, N DIV 8, accdesc] = value;
    return;
```

## Library pseudocode for aarch64/functions/memory/PhysMemTagRead

```
// PhysMemTagRead()
// =====
// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access from the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.

(PhysMemRetStatus, bits(4)) PhysMemTagRead(AddressDescriptor desc, AccessDescriptor accdesc);
```

## Library pseudocode for aarch64/functions/memory/PhysMemTagWrite

```
// PhysMemTagWrite()
// =====
// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access to the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.

PhysMemRetStatus PhysMemTagWrite(AddressDescriptor desc, AccessDescriptor accdesc, bits (4) value);
```

## Library pseudocode for aarch64/functions/memory/StoreOnlyTagCheckingEnabled

```
// StoreOnlyTagCheckingEnabled()
// =====
// Returns TRUE if loads executed at the given Exception level are Tag unchecked.

boolean StoreOnlyTagCheckingEnabled(bits (2) el)
    assert HaveMTEStoreOnlyExt();
    bit tcso;

    case el of
        when EL0
            if !ELIsInHost(el) then
                tcso = SCTLR_EL1.TCSO0;
            else
                tcso = SCTLR_EL2.TCSO0;
        when EL1
            tcso = SCTLR_EL1.TCSO;
        when EL2
            tcso = SCTLR_EL2.TCSO;
        otherwise
            tcso = SCTLR_EL3.TCSO;

    return tcso == '1';
```

## Library pseudocode for aarch64/functions/mops/CPYFOptionA

```
// CPYFOptionA()
// =====
// Returns TRUE if the implementation uses Option A for the
// CPYF* instructions, and FALSE otherwise.

boolean CPYFOptionA()
    return boolean IMPLEMENTATION_DEFINED "CPYF* instructions use Option A";
```

## Library pseudocode for aarch64/functions/mops/CPYOptionA

```
// CPYOptionA()
// =====
// Returns TRUE if the implementation uses Option A for the
// CPY* instructions, and FALSE otherwise.

boolean CPYOptionA()
    return boolean IMPLEMENTATION_DEFINED "CPY* instructions use Option A";
```

### Library pseudocode for aarch64/functions/mops/CPYPostSizeChoice

```
// CPYPostSizeChoice()
// =====
// Returns the size of the copy that is performed by the CPYE* instructions for this
// implementation given the parameters of the destination, source and size of the copy.
// Postsize is encoded as -1*size for an option A implementation if cpysize is negative.

bits(64) CPYPostSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);
```

### Library pseudocode for aarch64/functions/mops/CPYPreSizeChoice

```
// CPYPreSizeChoice()
// =====
// Returns the size of the copy that is performed by the CPYP* instructions for this
// implementation given the parameters of the destination, source and size of the copy.
// Presize is encoded as -1*size for an option A implementation if cpysize is negative.

bits(64) CPYPreSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);
```

### Library pseudocode for aarch64/functions/mops/CPYSizeChoice

```
// CPYSizeChoice()
// =====
// Returns the size of the block this performed for an iteration of the copy given the
// parameters of the destination, source and size of the copy.

integer CPYSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);
```

### Library pseudocode for aarch64/functions/mops/CheckMOPSEnabled

```
// CheckMOPSEnabled()
// =====
// Check for EL0 and EL1 access to the CPY* and SET* instructions.

CheckMOPSEnabled()
    if (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0') &&
        (!IsHCRXEL2Enabled() || HCRX_EL2.MSCEn == '0')) then
        UNDEFINED;

    if (PSTATE.EL == EL0 && SCTLRL_EL1.MSCEn == '0' &&
        (!EL2Enabled() || HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0')) then
        UNDEFINED;

    if PSTATE.EL == EL0 && IsInHost() && SCTLRL_EL2.MSCEn == '0' then
        UNDEFINED;
```

## Library pseudocode for aarch64/functions/mops/CheckMemCpyParams

```
// CheckMemCpyParams()
// =====
// Check if the parameters to a CPY* or CPYF* instruction are consistent with the
// PE state and well-formed.

CheckMemCpyParams(MOPSSStage stage, boolean implements_option_a, bits(4) nzcvc, bits(4) options,
                  integer d, integer s, integer n, bits(64) toaddress, bits(64) fromaddress,
                  bits(64) cpysize)
boolean from_epilogue = stage == MOPSSStage\_Epilogue;
// Check if this version is consistent with the state of the call.
if MemCpyZeroSizeCheck() || SInt(cpysize) != 0 then
    boolean using_option_a = nzcvc<1> == '0';
    if implements_option_a != using_option_a then
        boolean wrong_option = TRUE;
        MismatchedMemCpyException(implements_option_a, d, s, n, wrong_option,
                                from_epilogue, options);

// Check if the parameters to this instruction are valid.
if stage == MOPSSStage\_Main then
    if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
        boolean wrong_option = FALSE;
        MismatchedMemCpyException(implements_option_a, d, s, n, wrong_option,
                                from_epilogue, options);
else
    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
        boolean wrong_option = FALSE;
        MismatchedMemCpyException(implements_option_a, d, s, n, wrong_option,
                                from_epilogue, options);
```

## Library pseudocode for aarch64/functions/mops/CheckMemSetParams

```
// CheckMemSetParams()
// =====
// Check if the parameters to a SET* or SETG* instruction are consistent with the
// PE state and well-formed.

CheckMemSetParams(MOPSSStage stage, boolean implements_option_a, bits(4) nzcvc, bits(2) options,
                  integer d, integer s, integer n, bits(64) toaddress, bits(64) setsize,
                  boolean is_setg)
boolean from_epilogue = stage == MOPSSStage\_Epilogue;

// Check if this version is consistent with the state of the call.
if MemCpyZeroSizeCheck() || SInt(setsize) != 0 then
    boolean using_option_a = nzcvc<1> == '0';
    if implements_option_a != using_option_a then
        boolean wrong_option = TRUE;
        MismatchedMemSetException(implements_option_a, d, s, n, wrong_option,
                                from_epilogue, options, is_setg);

// Check if the parameters to this instruction are valid.
if stage == MOPSSStage\_Main then
    if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
        boolean wrong_option = FALSE;
        MismatchedMemSetException(implements_option_a, d, s, n, wrong_option,
                                from_epilogue, options, is_setg);
else
    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
    if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
        boolean wrong_option = FALSE;
        MismatchedMemSetException(implements_option_a, d, s, n, wrong_option,
                                from_epilogue, options, is_setg);
```

## Library pseudocode for aarch64/functions/mops/IsMemCpyForward

```
// IsMemCpyForward()
// =====
// Returns TRUE if in a memcpy of size cpysize bytes from the source address fromaddress
// to destination address toaddress is done in the forward direction on this implementation.

boolean IsMemCpyForward(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize)
    boolean forward;

    // Check for overlapping cases
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) &&
        (UInt(fromaddress<55:0>) < UInt(ZeroExtend(toaddress<55:0>, 64) + cpysize))) then
        forward = TRUE;

    elseif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) &&
            (UInt(ZeroExtend(fromaddress<55:0>, 64) + cpysize) > UInt(toaddress<55:0>))) then
        forward = FALSE;

    // Non-overlapping case
    else
        forward = boolean IMPLEMENTATION_DEFINED "CPY in the forward direction";

    return forward;
```

## Library pseudocode for aarch64/functions/mops/MOPSStage

```
// MOPSStage
// =====

enumeration MOPSStage { MOPSStage_Prologue, MOPSStage_Main, MOPSStage_Epilogue };
```

## Library pseudocode for aarch64/functions/mops/MaxBlockSizeCopiedBytes

```
// MaxBlockSizeCopiedBytes()
// =====
// Returns the maximum number of bytes that can used in a single block of the copy.

integer MaxBlockSizeCopiedBytes()
    return integer IMPLEMENTATION_DEFINED "Maximum bytes used in a single block of a copy";
```

## Library pseudocode for aarch64/functions/mops/MemCpyParametersIllformedE

```
// MemCpyParametersIllformedE()
// =====
// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPYE* instruction for this implementation given the parameters of the destination,
// source and size of the copy.

boolean MemCpyParametersIllformedE(bits(64) toaddress, bits(64) fromaddress,
                                    bits(64) cpysize);
```

## Library pseudocode for aarch64/functions/mops/MemCpyParametersIllformedM

```
// MemCpyParametersIllformedM()
// =====
// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPYM* instruction for this implementation given the parameters of the destination,
// source and size of the copy.

boolean MemCpyParametersIllformedM(bits(64) toaddress, bits(64) fromaddress,
                                    bits(64) cpysize);
```

## Library pseudocode for aarch64/functions/mops/MemCpyStageSize

```
// MemCpyStageSize()
// =====
// Returns the number of bytes copied by the given stage of a CPY* or CPYF* instruction.

bits(64) MemCpyStageSize(MOPSSStage stage, bits(64) toaddress, bits(64) fromaddress,
                        bits(64) cpysize)
    bits(64) stagecpysize;
    if stage == MOPSSStage_Prologue then
        // IMP DEF selection of the amount covered by pre-processing.
        stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
        assert stagecpysize<63> == cpysize<63> || IsZero(stagecpysize);

        if SInt(cpysize) > 0 then
            assert SInt(stagecpysize) <= SInt(cpysize);
        else
            assert SInt(stagecpysize) >= SInt(cpysize);
    else
        bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
        assert postsize<63> == cpysize<63> || IsZero(postsize);

        if stage == MOPSSStage_Main then
            stagecpysize = cpysize - postsize;
        else
            stagecpysize = postsize;

    return stagecpysize;
```

## Library pseudocode for aarch64/functions/mops/MemCpyZeroSizeCheck

```
// MemCpyZeroSizeCheck()
// =====
// Returns TRUE if the implementation option is checked on a copy of size zero remaining.

boolean MemCpyZeroSizeCheck();
```

## Library pseudocode for aarch64/functions/mops/MemSetParametersIllformedE

```
// MemSetParametersIllformedE()
// =====
// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETE* or SETGE* instruction for this implementation given the
// parameters of the destination and size of the set.

boolean MemSetParametersIllformedE(bits(64) toaddress, bits(64) setsize, boolean is_setge);
```

## Library pseudocode for aarch64/functions/mops/MemSetParametersIllformedM

```
// MemSetParametersIllformedM()
// =====
// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETM* or SETGM* instruction for this implementation given the
// parameters of the destination and size of the copy.

boolean MemSetParametersIllformedM(bits(64) toaddress, bits(64) setsize, boolean is_setgm);
```



## Library pseudocode for aarch64/functions/mops/MemSetStageSize

```
// MemSetStageSize()
// =====
// Returns the number of bytes set by the given stage of a SET* or SETG* instruction.

bits(64) MemSetStageSize(MOPSSStage stage, bits(64) toaddress, bits(64) setsize, boolean is_setg)
    bits(64) stagesetsize;
    if stage == MOPSSStage_Prologue then
        stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
        assert stagesetsize<63> == setsize<63> || IsZero(stagesetsize);
        if is_setg then assert stagesetsize<3:0> == '0000';

        if SInt(setsize) > 0 then
            assert SInt(stagesetsize) <= SInt(setsize);
        else
            assert SInt(stagesetsize) >= SInt(setsize);
    else
        bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
        assert postsize<63> == setsize<63> || IsZero(postsize);
        if is_setg then assert postsize<3:0> == '0000';

        if stage == MOPSSStage_Main then
            stagesetsize = setsize - postsize;
        else
            stagesetsize = postsize;

    return stagesetsize;
```

## Library pseudocode for aarch64/functions/mops/MemSetZeroSizeCheck

```
// MemSetZeroSizeCheck()
// =====
// Returns TRUE if the implementation option is checked on a copy of size zero remaining.

boolean MemSetZeroSizeCheck();
```

## Library pseudocode for aarch64/functions/mops/MismatchedCpySetTargetEL

```
// MismatchedCpySetTargetEL()
// =====
// Return the target exception level for an Exception_MemCpyMemSet.

bits(2) MismatchedCpySetTargetEL()
    bits(2) target_el;

    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    elsif (PSTATE.EL == EL1 && EL2Enabled() &&
        IsHCRXEL2Enabled() && HCRX_EL2.MCE2 == '1') then
        target_el = EL2;
    else
        target_el = EL1;

    return target_el;
```

## Library pseudocode for aarch64/functions/mops/MismatchedMemCpyException

```
// MismatchedMemCpyException()
// =====
// Generates an exception for a CPY* instruction if the version
// is inconsistent with the state of the call.

MismatchedMemCpyException(boolean option_a, integer destreg, integer srcreg, integer sizereg,
                          boolean wrong_option, boolean from_epilogue, bits(4) options)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;
    bits(2) target_el = MismatchedCpySetTargetEL();

    ExceptionRecord except = ExceptionSyndrome(Exception\_MemCpyMemSet);
    except.syndrome<24> = '0';
    except.syndrome<23> = '0';
    except.syndrome<22:19> = options;
    except.syndrome<18> = if from_epilogue then '1' else '0';
    except.syndrome<17> = if wrong_option then '1' else '0';
    except.syndrome<16> = if option_a then '1' else '0';
    // exception.syndrome<15> is RES0
    except.syndrome<14:10> = destreg<4:0>;
    except.syndrome<9:5> = srcreg<4:0>;
    except.syndrome<4:0> = sizereg<4:0>;

    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/functions/mops/MismatchedMemSetException

```
// MismatchedMemSetException()
// =====
// Generates an exception for a SET* instruction if the version
// is inconsistent with the state of the call.

MismatchedMemSetException(boolean option_a, integer destreg, integer datareg, integer sizereg,
                          boolean wrong_option, boolean from_epilogue, bits(2) options,
                          boolean is_SETG)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;
    bits(2) target_el = MismatchedCpySetTargetEL();

    ExceptionRecord except = ExceptionSyndrome(Exception\_MemCpyMemSet);
    except.syndrome<24> = '1';
    except.syndrome<23> = if is_SETG then '1' else '0';
    // exception.syndrome<22:21> is RES0
    except.syndrome<20:19> = options;
    except.syndrome<18> = if from_epilogue then '1' else '0';
    except.syndrome<17> = if wrong_option then '1' else '0';
    except.syndrome<16> = if option_a then '1' else '0';
    // exception.syndrome<15> is RES0
    except.syndrome<14:10> = destreg<4:0>;
    except.syndrome<9:5> = datareg<4:0>;
    except.syndrome<4:0> = sizereg<4:0>;

    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/functions/mops/SETGOptionA

```
// SETGOptionA()
// =====
// Returns TRUE if the implementation uses Option A for the
// SETG* instructions, and FALSE otherwise.

boolean SETGOptionA()
    return boolean IMPLEMENTATION_DEFINED "SETG* instructions use Option A";
```

### Library pseudocode for aarch64/functions/mops/SETOptionA

```
// SETOptionA()
// =====
// Returns TRUE if the implementation uses Option A for the
// SET* instructions, and FALSE otherwise.

boolean SETOptionA()
    return boolean IMPLEMENTATION_DEFINED "SET* instructions use Option A";
```

### Library pseudocode for aarch64/functions/mops/SETPostSizeChoice

```
// SETPostSizeChoice()
// =====
// Returns the size of the set that is performed by the SETE* or SETGE* instructions
// for this implementation, given the parameters of the destination and size of the set.
// Postsize is encoded as -1*size for an option A implementation if setsize is negative.

bits(64) SETPostSizeChoice(bits(64) toaddress, bits(64) setsize, boolean is_setge);
```

### Library pseudocode for aarch64/functions/mops/SETPreSizeChoice

```
// SETPreSizeChoice()
// =====
// Returns the size of the set that is performed by the SETP* or SETGP* instructions
// for this implementation, given the parameters of the destination and size of the set.
// Presize is encoded as -1*size for an option A implementation if setsize is negative.

bits(64) SETPreSizeChoice(bits(64) toaddress, bits(64) setsize, boolean is_setgp);
```

### Library pseudocode for aarch64/functions/mops/SETSizeChoice

```
// SETSizeChoice()
// =====
// Returns the size of the block thisperformed for an iteration of the set given
// the parameters of the destination and size of the set. The size of the block
// is an integer multiple of alignsize.

integer SETSizeChoice(bits(64) toaddress, bits(64) setsize, integer alignsize);
```

### Library pseudocode for aarch64/functions/movewideop/MoveWideOp

```
// MoveWideOp
// =====
// Move wide 16-bit immediate instruction types.

enumeration MoveWideOp {MoveWideOp_N, MoveWideOp_Z, MoveWideOp_K};
```

## Library pseudocode for aarch64/functions/movwpreferred/MoveWidePreferred

```
// MoveWidePreferred()
// =====
//
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<-ORR alias.

boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)
    integer s = UInt(imms);
    integer r = UInt(immr);
    integer width = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
    if sf == '1' && !((immN:imms) IN {'1xxxxxx'}) then
        return FALSE;
    if sf == '0' && !((immN:imms) IN {'00xxxxx'}) then
        return FALSE;

    // for MOVZ must contain no more than 16 ones
    if s < 16 then
        // ones must not span halfword boundary when rotated
        return (-r MOD 16) <= (15 - s);

    // for MOVN must contain no more than 16 zeros
    if s >= width - 15 then
        // zeros must not span halfword boundary when rotated
        return (r MOD 16) <= (s - (width - 15));

    return FALSE;
```



```

// AddPAC()
// =====
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

bits(64) AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)
    bits(64) PAC;
    bits(64) result;
    bits(64) ext_ptr;
    bits(64) extfield;
    bit selbit;
    boolean isgeneric = FALSE;
    boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    boolean mtX = EffectiveMTX(ptr, !data, PSTATE.EL) == '1';
    integer top_bit = if tbi then 55 else 63;

    // If tagged pointers are in use for a regime with two TTBRs, use bit<55> of
    // the pointer to select between upper and lower ranges, and preserve this.
    // This handles the awkward case where there is apparently no correct choice between
    // the upper and lower address range - ie an addr of 1xxxxxxx0... with TBI0=0 and TBI1=1
    // and 0xxxxxxx1 with TBI1=0 and TBI0=1:
    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            if data then
                if TCR_EL1.TBI1 == '1' || TCR_EL1.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
                    (TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
        else
            // EL2 translation regime registers
            if data then
                if TCR_EL2.TBI1 == '1' || TCR_EL2.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0') ||
                    (TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
        else selbit = if tbi then ptr<55> else ptr<63>;

    if HaveEnhancedPAC2() && ConstPACField() then selbit = ptr<55>;
    integer bottom_PAC_bit = CalculateBottomPACBit(selbit);

    // If the VA is 56 or 55 bits and Top Byte is Ignored,
    // there are no unused bits left to insert the PAC
    if tbi && bottom_PAC_bit >= 55 then
        return ptr;

    extfield = Replicate(selbit, 64);

    // Compute the pointer authentication code for a ptr with good extension bits
    if tbi then
        ext_ptr = (ptr<63:56> :
                    extfield<(56-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>);
    elsif mtX then
        ext_ptr = (extfield<63:60> : ptr<59:56> :
                    extfield<(56-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>);
    else
        ext_ptr = extfield<(64-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>;

```

```

PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>, isgeneric);

// Check if the ptr has good extension bits and corrupt the pointer authentication code if not
bits(64) unusedbits_mask = Zeros(64);
unusedbits_mask<54:bottom_PAC_bit> = Ones((54-bottom_PAC_bit)+1);
if tbi then
    unusedbits_mask<63:56> = Ones(8);
elseif mtX then
    unusedbits_mask<63:60> = Ones(4);
if !IsZero(ptr AND unusedbits_mask) && ((ptr AND unusedbits_mask) != unusedbits_mask) then
    if HaveEnhancedPAC() then
        PAC = 0x0000000000000000<63:0>;
    elseif !HaveEnhancedPAC2() then
        PAC<top_bit-1> = NOT(PAC<top_bit-1>);

// Preserve the determination between upper and lower address at bit<55> and insert PAC into
// bits that are not used for the address or the tag(s).
if !HaveEnhancedPAC2() then
    if tbi then
        result = ptr<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
    else
        result = PAC<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
        // A compliant implementation of FEAT_MTE4 also implements FEAT_PAuth2
        assert !mtX;
else
    if tbi then
        result = (ptr<63:56> : selbit :
            (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>) :
            ptr<bottom_PAC_bit-1:0>);
    elseif mtX then
        result = ((ptr<63:60> EOR PAC<63:60>) : ptr<59:56> : selbit :
            (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>) :
            ptr<bottom_PAC_bit-1:0>);
    else
        result = ((ptr<63:56> EOR PAC<63:56>) : selbit :
            (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>) :
            ptr<bottom_PAC_bit-1:0>);
return result;

```

## Library pseudocode for aarch64/functions/pac/addpacda/AddPACDA

```
// AddPACDA()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APDAKey_EL1.

bits(64) AddPACDA(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnDA else SCTLRL_EL2.EnDA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLRL_EL1.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLRL_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLRL_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return AddPAC(x, y, APDAKey_EL1, TRUE);
```



## Library pseudocode for aarch64/functions/pac/addpacdb/AddPACDB

```
// AddPACDB()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APDBKey_EL1.

bits(64) AddPACDB(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTL_EL1.EnDB else SCTL_EL2.EnDB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTL_EL1.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTL_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTL_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elseif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elseif TrapEL2 then
        TrapPACUse(EL2);
    elseif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return AddPAC(x, y, APDBKey_EL1, TRUE);
```

## Library pseudocode for aarch64/functions/pac/addpacga/AddPACGA

```
// AddPACGA()
// =====
// Returns a 64-bit value where the lower 32 bits are 0, and the upper 32 bits contain
// a 32-bit pointer authentication code which is derived using a cryptographic
// algorithm as a combination of x, y and the APGAKey_EL1.

bits(64) AddPACGA(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(128) APGAKey_EL1;
    boolean isgeneric = TRUE;

    APGAKey_EL1 = APGAKeyHi_EL1<63:0> : APGAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elseif TrapEL2 then
        TrapPACUse(EL2);
    elseif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return ComputePAC(x, y, APGAKey_EL1<127:64>, APGAKey_EL1<63:0>, isgeneric)<63:32>:Zeros(32);
```

## Library pseudocode for aarch64/functions/pac/addpacia/AddPACIA

```
// AddPACIA()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y, and the
// APIAKey_EL1.

bits(64) AddPACIA(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLRL_EL1.EnIA else SCTLRL_EL2.EnIA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLRL_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLRL_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLRL_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elseif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elseif TrapEL2 then
        TrapPACUse(EL2);
    elseif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return AddPAC(x, y, APIAKey_EL1, FALSE);
```

## Library pseudocode for aarch64/functions/pac/addpacib/AddPACIB

```
// AddPACIB()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APIBKey_EL1.

bits(64) AddPACIB(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elseif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elseif TrapEL2 then
        TrapPACUse(EL2);
    elseif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return AddPAC(x, y, APIBKey_EL1, FALSE);
```

## Library pseudocode for aarch64/functions/pac/auth/AArch64.PACFailException

```
// AArch64.PACFailException()
// =====
// Generates a PAC Fail Exception

AArch64.PACFailException(bits(2) syndrome)
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception\_PACFail);
    except.syndrome<1:0> = syndrome;
    except.syndrome<24:2> = Zeros(23); // RES0

    if UInt(PSTATE.EL) > UInt(EL0) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```



```

// Auth()
// =====
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit key_number,
              boolean is_combined)
    bits(64) PAC;
    bits(64) result;
    bits(64) original_ptr;
    bits(2) error_code;
    bits(64) extfield;
    boolean isgeneric = FALSE;

    // Reconstruct the extension field used of adding the PAC to the pointer
    boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    boolean mtx = EffectiveMTX(ptr, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = CalculateBottomPACBit(ptr<55>);
    extfield = Replicate(ptr<55>, 64);

    // If the VA is 56 or 55 bits and Top Byte is Ignored,
    // there are no unused bits left for the PAC
    if tbi && bottom_PAC_bit >= 55 then
        return ptr;

    if tbi then
        original_ptr = (ptr<63:56> :
                        extfield<(56-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>);
    elsif mtx then
        original_ptr = (extfield<63:60> : ptr<59:56> :
                        extfield<(56-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>);
    else
        original_ptr = extfield<(64-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(original_ptr, modifier, K<127:64>, K<63:0>, isgeneric);
    // Check pointer authentication code
    if tbi then
        if !HaveEnhancedPAC2() then
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63:55>;error_code:original_ptr<52:0>;
        else
            result = ptr;
            result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
                if result<54:bottom_PAC_bit> != Replicate(result<55>, (55-bottom_PAC_bit)) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
            elsif mtx then
                assert HaveEnhancedPAC2();
                result = ptr;
                result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
                result<63:60> = result<63:60> EOR PAC<63:60>;
                if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
                    if ((result<54:bottom_PAC_bit> != Replicate(result<55>, (55-bottom_PAC_bit))) ||
                        (result<63:60> != Replicate(result<55>, 4))) then
                        error_code = (if data then '1' else '0'):key_number;
                        AArch64.PACFailException(error_code);
            else
                if !HaveEnhancedPAC2() then
                    if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> && PAC<63:56> == ptr<63:56> then
                        result = original_ptr;
                    else
                        error_code = key_number:NOT(key_number);
                        result = original_ptr<63>;error_code:original_ptr<60:0>;

```

```

else
    result = ptr;
    result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
    result<63:56> = result<63:56> EOR PAC<63:56>;
    if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
        if result<63:bottom_PAC_bit> != Replicate(result<55>, (64-bottom_PAC_bit)) then
            error_code = (if data then '1' else '0'):key_number;
            AArch64.PACFailException(error_code);
return result;

```

## Library pseudocode for aarch64/functions/pac/authda/AuthDA

```

// AuthDA()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACDA().

```

```

bits(64) AuthDA(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return Auth(x, y, APDAKey_EL1, TRUE, '0', is_combined);

```



## Library pseudocode for aarch64/functions/pac/authdb/AuthDB

```
// AuthDB()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of x, using
// the same algorithm and key as AddPACDB().

bits(64) AuthDB(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTL_R_EL1.EnDB else SCTL_R_EL2.EnDB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTL_R_EL1.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTL_R_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTL_R_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return Auth(x, y, APDBKey_EL1, TRUE, '1', is_combined);
```

## Library pseudocode for aarch64/functions/pac/authia/AuthIA

```
// AuthIA()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIA().

bits(64) AuthIA(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return Auth(x, y, APIAKey_EL1, FALSE, '0', is_combined);
```

## Library pseudocode for aarch64/functions/pac/authib/AuthIB

```
// AuthIB()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIB().

bits(64) AuthIB(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return Auth(x, y, APIBKey_EL1, FALSE, '1', is_combined);
```

## Library pseudocode for aarch64/functions/pac/calcbottompacbit/AArch64.PACEffectiveTxSZ

```
// AArch64.PACEffectiveTxSZ()
// =====
// Compute the effective value for TxSZ used to determine the placement of the PAC field

bits(6) AArch64.PACEffectiveTxSZ(Regime regime, S1TTWParams walkparams)
    constant integer slmaxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    constant integer slmintxsz = AArch64.S1MinTxSZ(regime, walkparams.d128,
                                                walkparams.ds, walkparams.tgx);

    if AArch64.S1TxSZFaults(regime, walkparams) then
        if ConstrainUnpredictable(Unpredictable\_RESTnSZ) == Constraint\_FORCE then
            if UInt(walkparams.txsz) < slmintxsz then
                return slmintxsz<5:0>;
            if UInt(walkparams.txsz) > slmaxtxsz then
                return slmaxtxsz<5:0>;
        elseif UInt(walkparams.txsz) < slmintxsz then
            return slmintxsz<5:0>;
        elseif UInt(walkparams.txsz) > slmaxtxsz then
            return slmaxtxsz<5:0>;

    return walkparams.txsz;
```

## Library pseudocode for aarch64/functions/pac/calcbottompacbit/CalculateBottomPACBit

```
// CalculateBottomPACBit()
// =====

integer CalculateBottomPACBit(bit top_bit)
    Regime regime;
    S1TTWParams walkparams;
    integer bottom_PAC_bit;

    regime = TranslationRegime(PSTATE.EL);
    ss = CurrentSecurityState();
    walkparams = AArch64.GetS1TTWParams(regime, ss, Replicate(top_bit, 64));
    bottom_PAC_bit = 64 - UInt(AArch64.PACEffectiveTxSZ(regime, walkparams));

    return bottom_PAC_bit;
```

## Library pseudocode for aarch64/functions/pac/computepac/ComputePAC

```
// ComputePAC()
// =====

bits(64) ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1,
                    boolean isgeneric)
    if UsePACIMP(isgeneric) then
        return ComputePACIMPDEF(data, modifier, key0, key1);
    if UsePACQARMA3(isgeneric) then
        boolean isqarma3 = TRUE;
        return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
    if UsePACQARMA5(isgeneric) then
        boolean isqarma3 = FALSE;
        return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
```

## Library pseudocode for aarch64/functions/pac/computepac/ComputePACIMPDEF

```
// ComputePACIMPDEF()
// =====
// Compute IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.

bits(64) ComputePACIMPDEF(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1);
```



```

// ComputePACQARMA()
// =====
// Compute QARMA3 or QARMA5 cryptographic algorithm for PAC calculation

bits(64) ComputePACQARMA(bits(64) data, bits(64) modifier, bits(64) key0,
                        bits(64) key1, boolean isqarma3)

bits(64) workingval;
bits(64) runningmod;
bits(64) roundkey;
bits(64) modk0;
constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;

integer iterations;
RC[0] = 0x0000000000000000<63:0>;
RC[1] = 0x13198A2E03707344<63:0>;
RC[2] = 0xA4093822299F31D0<63:0>;

if isqarma3 then
    iterations = 2;
else // QARMA5
    iterations = 4;
    RC[3] = 0x082EFA98EC4E6C89<63:0>;
    RC[4] = 0x452821E638D01377<63:0>;

modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
runningmod = modifier;
workingval = data EOR key0;

for i = 0 to iterations
    roundkey = key1 EOR runningmod;
    workingval = workingval EOR roundkey;
    workingval = workingval EOR RC[i];
    if i > 0 then
        workingval = PACCellShuffle(workingval);
        workingval = PACMult(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACSub(workingval);
    runningmod = TweakShuffle(runningmod<63:0>);
roundkey = modk0 EOR runningmod;
workingval = workingval EOR roundkey;
workingval = PACCellShuffle(workingval);
workingval = PACMult(workingval);
if isqarma3 then
    workingval = PACSub1(workingval);
else
    workingval = PACSub(workingval);
workingval = PACCellShuffle(workingval);
workingval = PACMult(workingval);
workingval = key1 EOR workingval;
workingval = PACCellInvShuffle(workingval);
if isqarma3 then
    workingval = PACSub1(workingval);
else
    workingval = PACInvSub(workingval);
workingval = PACMult(workingval);
workingval = PACCellInvShuffle(workingval);
workingval = workingval EOR key0;
workingval = workingval EOR runningmod;
for i = 0 to iterations
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACInvSub(workingval);
    if i < iterations then
        workingval = PACMult(workingval);
        workingval = PACCellInvShuffle(workingval);
    runningmod = TweakInvShuffle(runningmod<63:0>);
    roundkey = key1 EOR runningmod;

```

```

        workingval = workingval EOR RC[iterations-i];
        workingval = workingval EOR roundkey;
        workingval = workingval EOR Alpha;
    workingval = workingval EOR modk0;

    return workingval;

```

#### Library pseudocode for aarch64/functions/pac/computepac/PACCellInvShuffle

```

// PACCellInvShuffle()
// =====

bits(64) PACCellInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<15:12>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<51:48>;
    outdata<15:12> = indata<39:36>;
    outdata<19:16> = indata<59:56>;
    outdata<23:20> = indata<47:44>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<19:16>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<31:28>;
    outdata<47:44> = indata<11:8>;
    outdata<51:48> = indata<23:20>;
    outdata<55:52> = indata<3:0>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = indata<63:60>;
    return outdata;

```

#### Library pseudocode for aarch64/functions/pac/computepac/PACCellShuffle

```

// PACCellShuffle()
// =====

bits(64) PACCellShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<55:52>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<47:44>;
    outdata<15:12> = indata<3:0>;
    outdata<19:16> = indata<31:28>;
    outdata<23:20> = indata<51:48>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<43:40>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<15:12>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = indata<23:20>;
    outdata<51:48> = indata<11:8>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<19:16>;
    outdata<63:60> = indata<63:60>;
    return outdata;

```

## Library pseudocode for aarch64/functions/pac/computepac/PACInvSub

```
// PACInvSub()
// =====

bits(64) PACInvSub(bits(64) Tinput)
// This is a 4-bit substitution from the PRINCE-family cipher
bits(64) Toutput;
for i = 0 to 15
    case Tinput<4*i+3:4*i> of
        when '0000' Toutput<4*i+3:4*i> = '0101';
        when '0001' Toutput<4*i+3:4*i> = '1110';
        when '0010' Toutput<4*i+3:4*i> = '1101';
        when '0011' Toutput<4*i+3:4*i> = '1000';
        when '0100' Toutput<4*i+3:4*i> = '1010';
        when '0101' Toutput<4*i+3:4*i> = '1011';
        when '0110' Toutput<4*i+3:4*i> = '0001';
        when '0111' Toutput<4*i+3:4*i> = '1001';
        when '1000' Toutput<4*i+3:4*i> = '0010';
        when '1001' Toutput<4*i+3:4*i> = '0110';
        when '1010' Toutput<4*i+3:4*i> = '1111';
        when '1011' Toutput<4*i+3:4*i> = '0000';
        when '1100' Toutput<4*i+3:4*i> = '0100';
        when '1101' Toutput<4*i+3:4*i> = '1100';
        when '1110' Toutput<4*i+3:4*i> = '0111';
        when '1111' Toutput<4*i+3:4*i> = '0011';
return Toutput;
```

## Library pseudocode for aarch64/functions/pac/computepac/PACMult

```
// PACMult()
// =====

bits(64) PACMult(bits(64) Sinput)
    bits(4) t0;
    bits(4) t1;
    bits(4) t2;
    bits(4) t3;
    bits(64) Soutput;

    for i = 0 to 3
        t0<3:0> = RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 2);
        t0<3:0> = t0<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t1<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        t1<3:0> = t1<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 2);
        t2<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 2) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1);
        t2<3:0> = t2<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t3<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 2);
        t3<3:0> = t3<3:0> EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        Soutput<4*i+3:4*i> = t3<3:0>;
        Soutput<4*(i+4)+3:4*(i+4)> = t2<3:0>;
        Soutput<4*(i+8)+3:4*(i+8)> = t1<3:0>;
        Soutput<4*(i+12)+3:4*(i+12)> = t0<3:0>;
    return Soutput;
```



### Library pseudocode for aarch64/functions/pac/computepac/PACSub

```
// PACSub()
// =====

bits(64) PACSub(bits(64) Tinput)
// This is a 4-bit substitution from the PRINCE-family cipher
bits(64) Toutput;
for i = 0 to 15
    case Tinput<4*i+3:4*i> of
        when '0000' Toutput<4*i+3:4*i> = '1011';
        when '0001' Toutput<4*i+3:4*i> = '0110';
        when '0010' Toutput<4*i+3:4*i> = '1000';
        when '0011' Toutput<4*i+3:4*i> = '1111';
        when '0100' Toutput<4*i+3:4*i> = '1100';
        when '0101' Toutput<4*i+3:4*i> = '0000';
        when '0110' Toutput<4*i+3:4*i> = '1001';
        when '0111' Toutput<4*i+3:4*i> = '1110';
        when '1000' Toutput<4*i+3:4*i> = '0011';
        when '1001' Toutput<4*i+3:4*i> = '0111';
        when '1010' Toutput<4*i+3:4*i> = '0100';
        when '1011' Toutput<4*i+3:4*i> = '0101';
        when '1100' Toutput<4*i+3:4*i> = '1101';
        when '1101' Toutput<4*i+3:4*i> = '0010';
        when '1110' Toutput<4*i+3:4*i> = '0001';
        when '1111' Toutput<4*i+3:4*i> = '1010';
return Toutput;
```

### Library pseudocode for aarch64/functions/pac/computepac/PacSub1

```
// PacSub1()
// =====

bits(64) PACSub1(bits(64) Tinput)
// This is a 4-bit substitution from Qarma sigma1
bits(64) Toutput;
for i = 0 to 15
    case Tinput<4*i+3:4*i> of
        when '0000' Toutput<4*i+3:4*i> = '1010';
        when '0001' Toutput<4*i+3:4*i> = '1101';
        when '0010' Toutput<4*i+3:4*i> = '1110';
        when '0011' Toutput<4*i+3:4*i> = '0110';
        when '0100' Toutput<4*i+3:4*i> = '1111';
        when '0101' Toutput<4*i+3:4*i> = '0111';
        when '0110' Toutput<4*i+3:4*i> = '0011';
        when '0111' Toutput<4*i+3:4*i> = '0101';
        when '1000' Toutput<4*i+3:4*i> = '1001';
        when '1001' Toutput<4*i+3:4*i> = '1000';
        when '1010' Toutput<4*i+3:4*i> = '0000';
        when '1011' Toutput<4*i+3:4*i> = '1100';
        when '1100' Toutput<4*i+3:4*i> = '1011';
        when '1101' Toutput<4*i+3:4*i> = '0001';
        when '1110' Toutput<4*i+3:4*i> = '0010';
        when '1111' Toutput<4*i+3:4*i> = '0100';
return Toutput;
```

### Library pseudocode for aarch64/functions/pac/computepac/RC

```
// RC[]
// ====

array bits(64) RC[0..4];
```

### Library pseudocode for aarch64/functions/pac/computepac/RotCell

```
// RotCell()
// =====

bits(4) RotCell(bits(4) incell, integer amount)
    bits(8) tmp;
    bits(4) outcell;

    // assert amount>3 || amount<1;
    tmp<7:0> = incell<3:0>:incell<3:0>;
    outcell = tmp<7-amount:4-amount>;
    return outcell;
```

### Library pseudocode for aarch64/functions/pac/computepac/TweakCellInvRot

```
// TweakCellInvRot()
// =====

bits(4) TweakCellInvRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<2>;
    outcell<2> = incell<1>;
    outcell<1> = incell<0>;
    outcell<0> = incell<0> EOR incell<3>;
    return outcell;
```

### Library pseudocode for aarch64/functions/pac/computepac/TweakCellRot

```
// TweakCellRot()
// =====

bits(4) TweakCellRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<0> EOR incell<1>;
    outcell<2> = incell<3>;
    outcell<1> = incell<2>;
    outcell<0> = incell<1>;
    return outcell;
```

### Library pseudocode for aarch64/functions/pac/computepac/TweakInvShuffle

```
// TweakInvShuffle()
// =====

bits(64) TweakInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = TweakCellInvRot(indata<51:48>);
    outdata<7:4> = indata<55:52>;
    outdata<11:8> = indata<23:20>;
    outdata<15:12> = indata<27:24>;
    outdata<19:16> = indata<3:0>;
    outdata<23:20> = indata<7:4>;
    outdata<27:24> = TweakCellInvRot(indata<11:8>);
    outdata<31:28> = indata<15:12>;
    outdata<35:32> = TweakCellInvRot(indata<31:28>);
    outdata<39:36> = TweakCellInvRot(indata<63:60>);
    outdata<43:40> = TweakCellInvRot(indata<59:56>);
    outdata<47:44> = TweakCellInvRot(indata<19:16>);
    outdata<51:48> = indata<35:32>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = TweakCellInvRot(indata<47:44>);
    return outdata;
```

### Library pseudocode for aarch64/functions/pac/computepac/TweakShuffle

```
// TweakShuffle()
// =====

bits(64) TweakShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<19:16>;
    outdata<7:4> = indata<23:20>;
    outdata<11:8> = TweakCellRot(indata<27:24>);
    outdata<15:12> = indata<31:28>;
    outdata<19:16> = TweakCellRot(indata<47:44>);
    outdata<23:20> = indata<11:8>;
    outdata<27:24> = indata<15:12>;
    outdata<31:28> = TweakCellRot(indata<35:32>);
    outdata<35:32> = indata<51:48>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = TweakCellRot(indata<63:60>);
    outdata<51:48> = TweakCellRot(indata<3:0>);
    outdata<55:52> = indata<7:4>;
    outdata<59:56> = TweakCellRot(indata<43:40>);
    outdata<63:60> = TweakCellRot(indata<39:36>);
    return outdata;
```

### Library pseudocode for aarch64/functions/pac/computepac/UsePACIMP

```
// UsePACIMP()
// =====
// Checks whether IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC
// calculation.

boolean UsePACIMP(boolean isgeneric)
    return if isgeneric then HavePACIMPGeneric() else HavePACIMPAuth();
```

### Library pseudocode for aarch64/functions/pac/computepac/UsePACQARMA3

```
// UsePACQARMA3()
// =====
// Checks whether QARMA3 cryptographic algorithm to be used for PAC calculation.

boolean UsePACQARMA3(boolean isgeneric)
    return if isgeneric then HavePACQARMA3Generic() else HavePACQARMA3Auth();
```

### Library pseudocode for aarch64/functions/pac/computepac/UsePACQARMA5

```
// UsePACQARMA5()
// =====
// Checks whether QARMA5 cryptographic algorithm to be used for PAC calculation.

boolean UsePACQARMA5(boolean isgeneric)
    return if isgeneric then HavePACQARMA5Generic() else HavePACQARMA5Auth();
```

### Library pseudocode for aarch64/functions/pac/pac/ConstPACField

```
// ConstPACField()
// =====
// Returns TRUE if bit<55> can be used to determine the size of the PAC field, FALSE otherwise.

boolean ConstPACField()
    return IsFeatureImplemented(FEAT_CONSTPACFIELD);
```

### Library pseudocode for aarch64/functions/pac/pac/HaveEnhancedPAC

```
// HaveEnhancedPAC()
// =====
// Returns TRUE if support for EnhancedPAC is implemented, FALSE otherwise.

boolean HaveEnhancedPAC()
    return IsFeatureImplemented(FEAT_EPAC);
```

### Library pseudocode for aarch64/functions/pac/pac/HaveEnhancedPAC2

```
// HaveEnhancedPAC2()
// =====
// Returns TRUE if support for EnhancedPAC2 is implemented, FALSE otherwise.

boolean HaveEnhancedPAC2()
    return IsFeatureImplemented(FEAT_PAuth2);
```

### Library pseudocode for aarch64/functions/pac/pac/HaveFPAC

```
// HaveFPAC()
// =====
// Returns TRUE if support for FPAC is implemented, FALSE otherwise.

boolean HaveFPAC()
    return IsFeatureImplemented(FEAT_FPAC);
```

### Library pseudocode for aarch64/functions/pac/pac/HaveFPACCombined

```
// HaveFPACCombined()
// =====
// Returns TRUE if support for FPACCombined is implemented, FALSE otherwise.

boolean HaveFPACCombined()
    return IsFeatureImplemented(FEAT_FPACCOMBINE);
```

### Library pseudocode for aarch64/functions/pac/pac/HavePACExt

```
// HavePACExt()
// =====
// Returns TRUE if support for the PAC extension is implemented, FALSE otherwise.

boolean HavePACExt()
    return IsFeatureImplemented(FEAT_PAuth);
```

### Library pseudocode for aarch64/functions/pac/pac/HavePACIMPAuth

```
// HavePACIMPAuth()
// =====
// Returns TRUE if support for PAC IMP Auth is implemented, FALSE otherwise.

boolean HavePACIMPAuth()
    return IsFeatureImplemented(FEAT_PACIMP);
```

### Library pseudocode for aarch64/functions/pac/pac/HavePACIMPGeneric

```
// HavePACIMPGeneric()
// =====
// Returns TRUE if support for PAC IMP Generic is implemented, FALSE otherwise.

boolean HavePACIMPGeneric()
    return IsFeatureImplemented(FEAT_PACIMP);
```

### Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA3Auth

```
// HavePACQARMA3Auth()
// =====
// Returns TRUE if support for PAC QARMA3 Auth is implemented, FALSE otherwise.

boolean HavePACQARMA3Auth()
    return IsFeatureImplemented(FEAT_PACQARMA3);
```

### Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA3Generic

```
// HavePACQARMA3Generic()
// =====
// Returns TRUE if support for PAC QARMA3 Generic is implemented, FALSE otherwise.

boolean HavePACQARMA3Generic()
    return IsFeatureImplemented(FEAT_PACQARMA3);
```

### Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA5Auth

```
// HavePACQARMA5Auth()
// =====
// Returns TRUE if support for PAC QARMA5 Auth is implemented, FALSE otherwise.

boolean HavePACQARMA5Auth()
    return IsFeatureImplemented(FEAT_PACQARMA5);
```

### Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA5Generic

```
// HavePACQARMA5Generic()
// =====
// Returns TRUE if support for PAC QARMA5 Generic is implemented, FALSE otherwise.

boolean HavePACQARMA5Generic()
    return IsFeatureImplemented(FEAT_PACQARMA5);
```

### Library pseudocode for aarch64/functions/pac/pac/PtrHasUpperAndLowerAddRanges

```
// PtrHasUpperAndLowerAddRanges()
// =====
// Returns TRUE if the pointer has upper and lower address ranges, FALSE otherwise.

boolean PtrHasUpperAndLowerAddRanges()
    regime = TranslationRegime(PSTATE.EL);
    return HasUnprivileged(regime);
```

## Library pseudocode for aarch64/functions/pac/strip/Strip

```
// Strip()
// =====
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
// code field bits with the extension of the address bits. This can apply to either
// instructions or data, where, as the use of tagged pointers is distinct, it might be
// handled differently.

bits(64) Strip(bits(64) A, boolean data)
    bits(64) original_ptr;
    bits(64) extfield;
    boolean tbi = EffectiveTBI(A, !data, PSTATE.EL) == '1';
    boolean mtX = EffectiveMTX(A, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = CalculateBottomPACBit(A<55>);
    extfield = Replicate(A<55>, 64);

    // If the VA is 56 or 55 bits and Top Byte is Ignored,
    // there are no unused bits left for the PAC
    if tbi && bottom_PAC_bit >= 55 then
        return A;

    if tbi then
        original_ptr = (A<63:56> :
                        extfield<(56-bottom_PAC_bit)-1:0> : A<bottom_PAC_bit-1:0>);
    elsif mtX then
        original_ptr = (extfield<63:60> : A<59:56> :
                        extfield<(56-bottom_PAC_bit)-1:0> : A<bottom_PAC_bit-1:0>);
    else
        original_ptr = extfield<(64-bottom_PAC_bit)-1:0> : A<bottom_PAC_bit-1:0>;

    return original_ptr;
```

## Library pseudocode for aarch64/functions/pac/trappacuse/TrapPACUse

```
// TrapPACUse()
// =====
// Used for the trapping of the pointer authentication functions by higher exception
// levels.

TrapPACUse(bits(2) target_el)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    ExceptionRecord except;
    vect_offset = 0;
    except = ExceptionSyndrome(Exception\_PACTrap);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/functions/predictionrestrict/AArch64.RestrictPrediction

```
// AArch64.RestrictPrediction()
// =====
// Clear all predictions in the context.

AArch64.RestrictPrediction(bits(64) val, RestrictType restriction)

    ExecutionCntxt c;
    target_el      = val<25:24>;

    // If the target EL is not implemented or the instruction is executed at an
    // EL lower than the specified level, the instruction is treated as a NOP.
    if !HaveEL(target_el) || UInt(target_el) > UInt(PSTATE.EL) then EndOfInstruction();

    bit ns  = val<26>;
    bit nse = val<27>;
    ss = TargetSecurityState(ns, nse);

    // If the combination of Security state and Exception level is not implemented,
    // the instruction is treated as a NOP.
    if ss == SS_Root && target_el != EL3 then EndOfInstruction();
    if !HaveRME() && target_el == EL3 && ss != SS_Secure then EndOfInstruction();

    c.security = ss;
    c.target_el = target_el;

    if EL2Enabled() then
        if (PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1 then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = FALSE;
            c.vmid          = VMID[];

            elsif (target_el == EL0 && !ELIsInHost(target_el)) || target_el == EL1 then
                c.is_vmid_valid = TRUE;
                c.all_vmid      = val<48> == '1';
                c.vmid          = val<47:32>; // Only valid if val<48> == '0';

            else
                c.is_vmid_valid = FALSE;
        else
            c.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = FALSE;
        c.asid          = ASID[];

        elsif target_el == EL0 then
            c.is_asid_valid = TRUE;
            c.all_asid      = val<16> == '1';
            c.asid          = val<15:0>; // Only valid if val<16> == '0';

        else
            c.is_asid_valid = FALSE;

    c.restriction = restriction;
    RESTRICT_PREDICTIONS(c);
```

## Library pseudocode for aarch64/functions/prefetch/Prefetch

```
// Prefetch()
// =====

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(5) prfop)
    PrefetchHint hint;
    integer target;
    boolean stream;

    case prfop<4:3> of
        when '00' hint = Prefetch\_READ;           // PLD: prefetch for load
        when '01' hint = Prefetch\_EXEC;           // PLI: preload instructions
        when '10' hint = Prefetch\_WRITE;          // PST: prepare for store
        when '11' return;                          // unallocated hint
    target = UInt(prfop<2:1>);                      // target cache level
    stream = (prfop<0> != '0');                    // streaming (non-temporal)
    Hint\_Prefetch(address, hint, target, stream);
    return;
```

## Library pseudocode for aarch64/functions/pstatefield/PSTATEField

```
// PSTATEField
// =====
// MSR (immediate) instruction destinations.

enumeration PSTATEField {PSTATEField_DAIFSet, PSTATEField_DAIFClr,
    PSTATEField_PAN, // Armv8.1
    PSTATEField_UAO, // Armv8.2
    PSTATEField_DIT, // Armv8.4
    PSTATEField_SSBS,
    PSTATEField_TCO, // Armv8.5
    PSTATEField_SVCRSM,
    PSTATEField_SVCRZA,
    PSTATEField_SVCRSMZA,
    PSTATEField_ALLINT,
    PSTATEField_PM,
    PSTATEField_SP
};
```



## Library pseudocode for aarch64/functions/ras/AArch64.ESBOperation

```
// AArch64.ESBOperation()
// =====
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64

AArch64.ESBOperation()
    bits(2) target_el;
    boolean masked;

    (masked, target_el) = AArch64.PhysicalErrorTarget();

    intdis = Halted() || ExternalDebugInterruptsDisabled(target_el);
    masked = masked || intdis;

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        // This function might be called for an interworking case, and INTdis is masking
        // the SError interrupt.
        if ELUsingAArch32\(S1TranslationRegime\)() then
            bits(32) syndrome = Zeros(32);
            syndrome<31> = '1'; // A
            syndrome<15:0> = AArch32.PhysicalErrorSyndrome();
            DISR = syndrome;
        else
            implicit_esb = FALSE;
            bits(64) syndrome = Zeros(64);
            syndrome<31> = '1'; // A
            syndrome<24:0> = AArch64.PhysicalErrorSyndrome(implicit_esb);
            DISR_EL1 = syndrome;
            ClearPendingPhysicalSError(); // Set ISR_EL1.A to 0

    return;
```

## Library pseudocode for aarch64/functions/ras/AArch64.EncodeAsyncErrorSyndrome

```
// AArch64.EncodeAsyncErrorSyndrome()
// =====
// Return the encoding for corresponding ErrorState.

bits(3) AArch64.EncodeAsyncErrorSyndrome(ErrorState errorstate)
    case errorstate of
        when ErrorState\_UC return '000';
        when ErrorState\_UEU return '001';
        when ErrorState\_UEO return '010';
        when ErrorState\_UER return '011';
        when ErrorState\_CE return '110';
        otherwise Unreachable();
```

## Library pseudocode for aarch64/functions/ras/AArch64.EncodeSyncErrorSyndrome

```
// AArch64.EncodeSyncErrorSyndrome()
// =====
// Return the encoding for corresponding ErrorState.

bits(2) AArch64.EncodeSyncErrorSyndrome(ErrorState errorstate)
    case errorstate of
        when ErrorState\_UC return '10';
        when ErrorState\_UEO return '11';
        when ErrorState\_UER return '00';
        otherwise Unreachable();
```

## Library pseudocode for aarch64/functions/ras/AArch64.PhysicalErrorSyndrome

```
// AArch64.PhysicalErrorSyndrome()
// =====
// Generate SError syndrome.

bits(25) AArch64.PhysicalErrorSyndrome(boolean implicit_esb)
    bits(25) syndrome = Zeros(25);
    FaultRecord fault = GetPendingPhysicalError();
    ErrorState errorstate = AArch64.PEErrorState(fault);
    if errorstate == ErrorState\_Uncategorized then
        syndrome = Zeros(25);
    elseif errorstate == ErrorState\_IMPDEF then
        syndrome<24> = '1'; // IDS
        syndrome<23:0> = bits(24) IMPLEMENTATION_DEFINED "IMPDEF ErrorState";
    else
        syndrome<24> = '0'; // IDS
        syndrome<13> = (if implicit_esb then '1' else '0'); // IESB
        syndrome<12:10> = AArch64.EncodeAsyncErrorSyndrome(errorstate); // AET
        syndrome<9> = fault.extflag; // EA
        syndrome<5:0> = '010001'; // DFSC
    return syndrome;
```

## Library pseudocode for aarch64/functions/ras/AArch64.vESBOperation

```
// AArch64.vESBOperation()
// =====
// Perform the AArch64 ESB operation for virtual SError interrupts, either for ESB
// executed in AArch64 state, or for ESB in AArch32 state with EL2 using AArch64 state

AArch64.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // If physical SError interrupts are routed to EL2, and TGE is not set, then a virtual
    // SError interrupt might be pending
    vSEI_enabled = HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR_EL2.VSE == '1';
    vintdis = Halted() || ExternalDebugInterruptsDisabled(EL1);
    vmasked = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        // This function might be called for the interworking case, and INTdis is masking
        // the virtual SError interrupt.
        if ELUsingAArch32(EL1) then
            bits(32) target = Zeros(32);
            target<31> = '1'; // A
            target<15:14> = VDFSR<15:14>; // AET
            target<12> = VDFSR<12>; // ExT
            target<9> = TTBCR.EAE; // LPAE
            if TTBCR.EAE == '1' then // Long-descriptor format
                target<5:0> = '010001'; // STATUS
            else // Short-descriptor format
                target<10,3:0> = '10110'; // FS
            VDISR = target;
        else
            bits(64) target = Zeros(64);
            target<31> = '1'; // A
            target<24:0> = VSESR_EL2<24:0>;
            VDISR_EL2 = target;
            HCR_EL2.VSE = '0'; // Clear pending virtual SError

    return;
```

### Library pseudocode for aarch64/functions/ras/FirstRecordOfNode

```
// FirstRecordOfNode()
// =====
// Return the first record in the node that contains the record n.

integer FirstRecordOfNode(integer n)
    for q = n downto 0
        if IsFirstRecordOfNode(q) then return q;
    Unreachable();
```

### Library pseudocode for aarch64/functions/ras/IsCommonFaultInjectionImplemented

```
// IsCommonFaultInjectionImplemented()
// =====
// Check if the Common Fault Injection Model Extension is implemented by the node that owns this
// error record.

boolean IsCommonFaultInjectionImplemented(integer n);
```

### Library pseudocode for aarch64/functions/ras/IsCountableErrorsRecorded

```
// IsCountableErrorsRecorded()
// =====
// Check whether Error record n records countable errors.

boolean IsCountableErrorsRecorded(integer n);
```

### Library pseudocode for aarch64/functions/ras/IsErrorAddressIncluded

```
// IsErrorAddressIncluded()
// =====
// Check whether Error record n includes an address associated with an error.

boolean IsErrorAddressIncluded(integer n);
```

### Library pseudocode for aarch64/functions/ras/IsErrorRecordImplemented

```
// IsErrorRecordImplemented()
// =====
// Is the error record n implemented

boolean IsErrorRecordImplemented(integer n);
```

### Library pseudocode for aarch64/functions/ras/IsFirstRecordOfNode

```
// IsFirstRecordOfNode()
// =====
// Check if the record q is the first error record in its node.

boolean IsFirstRecordOfNode(integer q);
```

### Library pseudocode for aarch64/functions/ras/IsSPMUCounterImplemented

```
// IsSPMUCounterImplemented()
// =====
// Does the System PMU s implement the counter n.

boolean IsSPMUCounterImplemented(integer s, integer n);
```

## Library pseudocode for aarch64/functions/rcw/ProtectionEnabled

```
// ProtectionEnabled()
// =====
// Returns TRUE if the ProtectedBit is
// enabled in the current Exception level.

boolean ProtectionEnabled(bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));
    if (!IsD128Enabled(el)) then
        case regime of
            when EL1
                return IsTCR2EL1Enabled() && TCR2_EL1.PnCH == '1';
            when EL2
                return IsTCR2EL2Enabled() && TCR2_EL2.PnCH == '1';
            when EL3
                return TCR_EL3.PnCH == '1';
    else
        return TRUE;
    return FALSE;
```

## Library pseudocode for aarch64/functions/rcw/RCW128\_PROTECTED\_BIT

```
constant integer RCW128_PROTECTED_BIT = 114;
```

## Library pseudocode for aarch64/functions/rcw/RCW64\_PROTECTED\_BIT

```
constant integer RCW64_PROTECTED_BIT = 52;
```



```

// RCWCheck()
// =====
// Returns nzcw based on : if the new value for RCW/RCWS instructions satisfy RCW and/or RCWS checks
// Z is set to 1 if RCW checks fail
// C is set to 0 if RCWS checks fail

bits(4) RCWCheck(bits(N) old, bits(N) new, boolean soft)
    assert N IN {64,128};
    integer protectedbit = if N == 128 then RCW128\_PROTECTED\_BIT else RCW64\_PROTECTED\_BIT;
    boolean rcw_fail = FALSE;
    boolean rcws_fail = FALSE;
    boolean rcw_state_fail = FALSE;
    boolean rcws_state_fail = FALSE;
    boolean rcw_mask_fail = FALSE;
    boolean rcws_mask_fail = FALSE;

    //Effective RCWMask calculation
    bits(N) rcwmask = RCWMASK_EL1<N-1:0>;
    if N == 64 then
        rcwmask<49:18> = Replicate(rcwmask<17>,32);
        rcwmask<0> = '0';
    else
        rcwmask<55:17> = Replicate(rcwmask<16>,39);
        rcwmask<126:125,120:119,107:101,90:56,1:0> = Zeros(48);

    //Effective RCWSMask calculation
    bits(N) rcwsoftmask = RCWSMASK_EL1<N-1:0>;
    if N == 64 then
        rcwsoftmask<49:18> = Replicate(rcwsoftmask<17>,32);
        rcwsoftmask<0> = '0';
        if(ProtectionEnabled(PSTATE.EL)) then
            rcwsoftmask<52> = '0';
    else
        rcwsoftmask<55:17> = Replicate(rcwsoftmask<16>,39);
        rcwsoftmask<126:125,120:119,107:101,90:56,1:0> = Zeros(48);
        rcwsoftmask<114> = '0';

    //RCW Checks
    //State Check
    if (ProtectionEnabled(PSTATE.EL)) then
        if old<protectedbit> == '1' then
            rcw_state_fail = new<protectedbit,0> != old<protectedbit,0>;
        elsif old<protectedbit> == '0' then
            rcw_state_fail = new<protectedbit> != old<protectedbit>;

    //Mask Check
    if (ProtectionEnabled(PSTATE.EL)) then
        if old<protectedbit,0> == '11' then
            rcw_mask_fail = IsZero((new EOR old) AND NOT(rcwmask));

    //RCWS Checks
    if soft then
        //State Check
        if old<0> == '1' then
            rcws_state_fail = new<0> != old<0>;
        elsif (!ProtectionEnabled(PSTATE.EL) ||
            (ProtectionEnabled(PSTATE.EL) && old<protectedbit> == '0')) then
            rcws_state_fail = new<0> != old<0>;
        //Mask Check
        if old<0> == '1' then
            rcws_mask_fail = IsZero((new EOR old) AND NOT(rcwsoftmask));

    rcw_fail = rcw_state_fail || rcw_mask_fail ;
    rcws_fail = rcws_state_fail || rcws_mask_fail;

    bit n = '0';
    bit z = if rcw_fail then '1' else '0';
    bit c = if rcws_fail then '0' else '1';
    bit v = '0';
    return <n, z, c, v>;

```

## Library pseudocode for aarch64/functions/reduceop/Reduce

```
// Reduce()
// =====

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)
    boolean altfp = HaveAltFP() && !UsingAArch32() && FPCR.AH == '1';
    return Reduce(op, input, esize, altfp);

// Reduce()
// =====
// Perform the operation 'op' on pairs of elements from the input vector,
// reducing the vector to a scalar result. The 'altfp' argument controls
// alternative floating-point behavior.

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize, boolean altfp)
    assert esize IN {8,16,32,64};
    integer half;
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;

    if N == esize then
        return input<esize-1:0>;

    half = N DIV 2;
    hi = Reduce(op, input<N-1:half>, esize, altfp);
    lo = Reduce(op, input<half-1:0>, esize, altfp);

    case op of
        when ReduceOp_FMINNUM
            result = FMinNum(lo, hi, FPCR[]);
        when ReduceOp_FMAXNUM
            result = FMaxNum(lo, hi, FPCR[]);
        when ReduceOp_FMIN
            result = FMin(lo, hi, FPCR[], altfp);
        when ReduceOp_FMAX
            result = FMax(lo, hi, FPCR[], altfp);
        when ReduceOp_FADD
            result = FAdd(lo, hi, FPCR[]);
        when ReduceOp_ADD
            result = lo + hi;

    return result;
```

## Library pseudocode for aarch64/functions/reduceop/ReduceOp

```
// ReduceOp
// =====
// Vector reduce instruction types.

enumeration ReduceOp {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,
    ReduceOp_FMIN, ReduceOp_FMAX,
    ReduceOp_FADD, ReduceOp_ADD};
```

## Library pseudocode for aarch64/functions/registers/AArch64.MaybeZeroRegisterUppers

```
// AArch64.MaybeZeroRegisterUppers()
// =====
// On taking an exception to AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero.

AArch64.MaybeZeroRegisterUppers()
    assert UsingAArch32();          // Always called from AArch32 state before entering AArch64 state

    integer first;
    integer last;
    boolean include_R15;
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        first = 0; last = 14; include_R15 = FALSE;
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) then
        first = 0; last = 30; include_R15 = FALSE;
    else
        first = 0; last = 30; include_R15 = TRUE;

    for n = first to last
        if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpredictable\_ZEROUPPER) then
            _R[n]<63:32> = Zeros(32);

    return;
```

## Library pseudocode for aarch64/functions/registers/AArch64.ResetGeneralRegisters

```
// AArch64.ResetGeneralRegisters()
// =====

AArch64.ResetGeneralRegisters()

    for i = 0 to 30
        X[i, 64] = bits(64) UNKNOWN;

    return;
```

## Library pseudocode for aarch64/functions/registers/AArch64.ResetSIMDFPRegisters

```
// AArch64.ResetSIMDFPRegisters()
// =====

AArch64.ResetSIMDFPRegisters()

    for i = 0 to 31
        V[i, 128] = bits(128) UNKNOWN;

    return;
```



## Library pseudocode for aarch64/functions/registers/AArch64.ResetSpecialRegisters

```
// AArch64.ResetSpecialRegisters()
// =====

AArch64.ResetSpecialRegisters()

    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;
    SPSR_EL1 = bits(64) UNKNOWN;
    ELR_EL1 = bits(64) UNKNOWN;
    if HaveEL(EL2) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(64) UNKNOWN;
        ELR_EL2 = bits(64) UNKNOWN;
    if HaveEL(EL3) then
        SP_EL3 = bits(64) UNKNOWN;
        SPSR_EL3 = bits(64) UNKNOWN;
        ELR_EL3 = bits(64) UNKNOWN;

    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL(EL1) then
        SPSR_fiq<31:0> = bits(32) UNKNOWN;
        SPSR_irq<31:0> = bits(32) UNKNOWN;
        SPSR_abt<31:0> = bits(32) UNKNOWN;
        SPSR_und<31:0> = bits(32) UNKNOWN;

    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    return;
```

## Library pseudocode for aarch64/functions/registers/AArch64.ResetSystemRegisters

```
// AArch64.ResetSystemRegisters()
// =====

AArch64.ResetSystemRegisters(boolean cold_reset);
```

## Library pseudocode for aarch64/functions/registers/PC

```
// PC - non-assignment form
// =====
// Read program counter.

bits(64) PC[]
    return _PC;
```

## Library pseudocode for aarch64/functions/registers/SP

```
// SP[] - assignment form
// =====
// Write to stack pointer from a 64-bit value.

SP[] = bits(64) value
    if PSTATE.SP == '0' then
        SP_EL0 = value;
    else
        case PSTATE.EL of
            when EL0 SP_EL0 = value;
            when EL1 SP_EL1 = value;
            when EL2 SP_EL2 = value;
            when EL3 SP_EL3 = value;
        return;

// SP[] - non-assignment form
// =====
// Read stack pointer with slice of 64 bits.

bits(64) SP[]
    if PSTATE.SP == '0' then
        return SP_EL0;
    else
        case PSTATE.EL of
            when EL0 return SP_EL0;
            when EL1 return SP_EL1;
            when EL2 return SP_EL2;
            when EL3 return SP_EL3;
```

## Library pseudocode for aarch64/functions/registers/SPMCFGR\_EL1

```
// SPMCFGR_EL1[] - non-assignment form
// =====
// Read the current configuration of System Performance monitor for
// System PMU 's'.

bits(64) SPMCFGR_EL1[integer s];
```

## Library pseudocode for aarch64/functions/registers/SPMCGCR\_EL1

```
// SPMCGCR_EL1[] - non-assignment form
// =====
// Read counter group 'n' configuration for System PMU 's'.

bits(64) SPMCGCR_EL1[integer s, integer n];
```

## Library pseudocode for aarch64/functions/registers/SPMCNTENCLR\_EL0

```
// SPMCNTENCLR_EL0[] - non-assignment form
// =====
// Read the current mapping of disabled event counters for an 's'.

bits(64) SPMCNTENCLR_EL0[integer s];

// SPMCNTENCLR_EL0[] - assignment form
// =====
// Disable event counters for System PMU 's'.

SPMCNTENCLR_EL0[integer s] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMCNTENSET\_EL0

```
// SPMCNTENSET_EL0[] - non-assignment form
// =====
// Read the current mapping for enabled event counters of System PMU 's'.

bits(64) SPMCNTENSET_EL0[integer s];

// SPMCNTENSET_EL0[] - assignment form
// =====
// Enable event counters of System PMU 's'.

SPMCNTENSET_EL0[integer s] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMCR\_EL0

```
// SPMCR_EL0[] - non-assignment form
// =====
// Read the control register for System PMU 's'.

bits(64) SPMCR_EL0[integer s];

// SPMCR_EL0[] - assignment form
// =====
// Write to the control register for System PMU 's'.

SPMCR_EL0[integer s] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMDEVAFF\_EL1

```
// SPMDEVAFF_EL1[] - non-assignment form
// =====
// Read the discovery information for System PMU 's'.

bits(64) SPMDEVAFF_EL1[integer s];
```

### Library pseudocode for aarch64/functions/registers/SPMDEVARCH\_EL1

```
// SPMDEVARCH_EL1[] - non-assignment form
// =====
// Read the discovery information for System PMU 's'.

bits(64) SPMDEVARCH_EL1[integer s];
```

### Library pseudocode for aarch64/functions/registers/SPMEVCNTR\_EL0

```
// SPMEVCNTR_EL0[] - non-assignment form
// =====
// Read a System PMU Event Counter register for counter 'n' of a given
// System PMU 's'.

bits(64) SPMEVCNTR_EL0[integer s, integer n];

// SPMEVCNTR_EL0[] - assignment form
// =====
// Write to a System PMU Event Counter register for counter 'n' of a given
// System PMU 's'.

SPMEVCNTR_EL0[integer s, integer n] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMEVFILT2R\_EL0

```
// SPMEVFILT2R_EL0[] - non-assignment form
// =====
// Read the additional event selection controls for
// counter 'n' of a given System PMU 's'.

bits(64) SPMEVFILT2R_EL0[integer s, integer n];

// SPMEVFILT2R_EL0[] - assignment form
// =====
// Configure the additional event selection controls for
// counter 'n' of a given System PMU 's'.

SPMEVFILT2R_EL0[integer s, integer n] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMEVFILTR\_EL0

```
// SPMEVFILTR_EL0[] - non-assignment form
// =====
// Read the additional event selection controls for
// counter 'n' of a given System PMU 's'.

bits(64) SPMEVFILTR_EL0[integer s, integer n];

// SPMEVFILTR_EL0[] - assignment form
// =====
// Configure the additional event selection controls for
// counter 'n' of a given System PMU 's'.

SPMEVFILTR_EL0[integer s, integer n] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMEVTYPER\_EL0

```
// SPMEVTYPER_EL0[] - non-assignment form
// =====
// Read the current mapping of event with event counter SPMEVCNTR_EL0
// for counter 'n' of a given System PMU 's'.

bits(64) SPMEVTYPER_EL0[integer s, integer n];

// SPMEVTYPER_EL0[] - assignment form
// =====
// Configure which event increments the event counter SPMEVCNTR_EL0, for
// counter 'n' of a given System PMU 's'.

SPMEVTYPER_EL0[integer s, integer n] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMIIDR\_EL1

```
// SPMIIDR_EL1[] - non-assignment form
// =====
// Read the discovery information for System PMU 's'.

bits(64) SPMIIDR_EL1[integer s];
```

### Library pseudocode for aarch64/functions/registers/SPMINTENCLR\_EL1

```
// SPMINTENCLR_EL1[] - non-assignment form
// =====
// Read the masking information for interrupt requests on overflows of
// implemented counters of System PMU 's'.

bits(64) SPMINTENCLR_EL1[integer s];

// SPMINTENCLR_EL1[] - assignment form
// =====
// Disable the generation of interrupt requests on overflows of
// implemented counters of System PMU 's'.

SPMINTENCLR_EL1[integer s] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMINTENSET\_EL1

```
// SPMINTENSET_EL1[] - non-assignment form
// =====
// Read the masking information for interrupt requests on overflows of
// implemented counters of System PMU 's'.

bits(64) SPMINTENSET_EL1[integer s];

// SPMINTENSET_EL1[] - assignment form
// =====
// Disable the generation of interrupt requests on overflows of
// implemented counters for System PMU 's'.

SPMINTENSET_EL1[integer s] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMOVSCCLR\_EL0

```
// SPMOVSCCLR_EL0[] - non-assignment form
// =====
// Read the overflow bit clear status of implemented counters for System PMU 's'.

bits(64) SPMOVSCCLR_EL0[integer s];

// SPMOVSCCLR_EL0[] - assignment form
// =====
// Clear the overflow bit clear status of implemented counters for
// System PMU 's'.

SPMOVSCCLR_EL0[integer s] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMOVSSSET\_EL0

```
// SPMOVSSSET_EL0[] - non-assignment form
// =====
// Read state of the overflow bit for the implemented event counters
// of System PMU 's'.

bits(64) SPMOVSSSET_EL0[integer s];

// SPMOVSSSET_EL0[] - assignment form
// =====
// Sets the state of the overflow bit for the implemented event counters
// of System PMU 's'.

SPMOVSSSET_EL0[integer s] = bits(64) value;
```

## Library pseudocode for aarch64/functions/registers/SPMROOTCR\_EL3

```
// SPMROOTCR_EL3[] - non-assignment form
// =====
// Read the observability of Root and Realm events by System Performance
// Monitor for System PMU 's'.

bits(64) SPMROOTCR_EL3[integer s];

// SPMROOTCR_EL3[] - assignment form
// =====
// Configure the observability of Root and Realm events by System
// Performance Monitor for System PMU 's'.

SPMROOTCR_EL3[integer s] = bits(64) value;
```

## Library pseudocode for aarch64/functions/registers/SPMSCR\_EL1

```
// SPMSCR_EL1[] - non-assignment form
// =====
// Read the observability of Secure events by System Performance Monitor
// for System PMU 's'.

bits(64) SPMSCR_EL1[integer s];

// SPMSCR_EL1[] - assignment form
// =====
// Configure the observability of secure events by System Performance
// Monitor for System PMU 's'.

SPMSCR_EL1[integer s] = bits(64) value;
```

## Library pseudocode for aarch64/functions/registers/V

```
// V[] - assignment form
// =====
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits.

V[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    integer vlen = if IsSVEEnabled(PSTATE.EL) then CurrentVL else 128;
    if ConstrainUnpredictableBool(Unpredictable\_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(value, MAX\_VL);
    else
        _Z[n]<vlen-1:0> = ZeroExtend(value, vlen);

// V[] - non-assignment form
// =====
// Read from SIMD&FP register with implicit slice of 8, 16
// 32, 64 or 128 bits.

bits(width) V[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    return _Z[n]<width-1:0>;
```

## Library pseudocode for aarch64/functions/registers/Vpart

```
// Vpart[] - non-assignment form
// =====
// Reads a 128-bit SIMD&FP register in up to two parts:
// part 0 returns the bottom 8, 16, 32 or 64 bits of a value held in the register;
// part 1 returns the top half of the bottom 64 bits or the top half of the 128-bit
// value held in the register.

bits(width) Vpart(integer n, integer part, integer width)
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width < 128;
        return V[n, width];
    else
        assert width IN {32,64};
        bits(128) vreg = V[n, 128];
        return vreg<(width * 2)-1:width>;

// Vpart[] - assignment form
// =====
// Writes a 128-bit SIMD&FP register in up to two parts:
// part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register;
// part 1 inserts a 64-bit value into the top half of the register.

Vpart(integer n, integer part, integer width) = bits(width) value
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width < 128;
        V[n, width] = value;
    else
        assert width == 64;
        bits(64) vreg = V[n, 64];
        V[n, 128] = value<63:0> : vreg;
```

## Library pseudocode for aarch64/functions/registers/X

```
// X[] - assignment form
// =====
// Write to general-purpose register from either a 32-bit or a 64-bit value,
// where the size of the value is passed as an argument.

X(integer n, integer width) = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then
        _R[n] = ZeroExtend(value, 64);
    return;

// X[] - non-assignment form
// =====
// Read from general-purpose register with an explicit slice of 8, 16, 32 or 64 bits.

bits(width) X(integer n, integer width)
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<width-1:0>;
    else
        return Zeros(width);
```

### Library pseudocode for aarch64/functions/shiftreg/DecodeShift

```
// DecodeShift()
// =====
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
    case op of
        when '00' return ShiftType_LSL;
        when '01' return ShiftType_LSR;
        when '10' return ShiftType_ASR;
        when '11' return ShiftType_ROR;
```

### Library pseudocode for aarch64/functions/shiftreg/ShiftReg

```
// ShiftReg()
// =====
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType shifttype, integer amount, integer N)
    bits(N) result = X[reg, N];
    case shifttype of
        when ShiftType_LSL result = LSL(result, amount);
        when ShiftType_LSR result = LSR(result, amount);
        when ShiftType_ASR result = ASR(result, amount);
        when ShiftType_ROR result = ROR(result, amount);
    return result;
```

### Library pseudocode for aarch64/functions/shiftreg/ShiftType

```
// ShiftType
// =====
// AArch64 register shifts.

enumeration ShiftType {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};
```



## Library pseudocode for aarch64/functions/sme/CounterToPredicate

```
// CounterToPredicate()
// =====

bits(width) CounterToPredicate(bits(16) pred, integer width)
    integer count;
    integer esize;
    integer elements;
    constant integer VL = CurrentVL;
    constant integer PL = VL DIV 8;
    integer maxbit = HighestSetBit(CeilPow2(PL * 4)<15:0>);
    assert maxbit <= 14;
    bits(PL*4) result;
    boolean invert = pred<15> == '1';

    assert width == PL || width == PL*2 || width == PL*3 || width == PL*4;

    if IsZero(pred<3:0>) then
        return Zeros(width);

    case pred<3:0> of
        when 'xxx1'
            count = UInt(pred<maxbit:1>);
            esize = 8;
        when 'xx10'
            count = UInt(pred<maxbit:2>);
            esize = 16;
        when 'x100'
            count = UInt(pred<maxbit:3>);
            esize = 32;
        when '1000'
            count = UInt(pred<maxbit:4>);
            esize = 64;

    elements = (VL * 4) DIV esize;
    result = Zeros(PL*4);
    constant integer psize = esize DIV 8;
    for e = 0 to elements-1
        bit pbit = if e < count then '1' else '0';
        if invert then
            pbit = NOT(pbit);
        Elem[result, e, psize] = ZeroExtend(pbit, psize);

    return result<width-1:0>;
```

## Library pseudocode for aarch64/functions/sme/EncodePredCount

```
// EncodePredCount()
// =====

bits(width) EncodePredCount(integer esize, integer elements,
                             integer count_in, boolean invert_in, integer width)
    integer count = count_in;
    boolean invert = invert_in;
    constant integer PL = CurrentVL DIV 8;
    assert width == PL;
    assert esize IN {8, 16, 32, 64};
    assert count >= 0 && count <= elements;
    bits(16) pred;

    if count == 0 then
        return Zeros(width);

    if invert then
        count = elements - count;
    elsif count == elements then
        count = 0;
        invert = TRUE;

    bit inv = (if invert then '1' else '0');
    case esize of
        when 8  pred = inv : count<13:0> : '1';
        when 16 pred = inv : count<12:0> : '10';
        when 32 pred = inv : count<11:0> : '100';
        when 64 pred = inv : count<10:0> : '1000';

    return ZeroExtend(pred, width);
```

## Library pseudocode for aarch64/functions/sme/HaveSME

```
// HaveSME()
// =====
// Returns TRUE if the SME extension is implemented, FALSE otherwise.

boolean HaveSME()
    return IsFeatureImplemented(FEAT_SME);
```

## Library pseudocode for aarch64/functions/sme/HaveSME2

```
// HaveSME2()
// =====
// Returns TRUE if the SME2 extension is implemented, FALSE otherwise.

boolean HaveSME2()
    return IsFeatureImplemented(FEAT_SME2);
```

## Library pseudocode for aarch64/functions/sme/HaveSME2p1

```
// HaveSME2p1()
// =====
// Returns TRUE if the SME2.1 extension is implemented, FALSE otherwise.

boolean HaveSME2p1()
    return IsFeatureImplemented(FEAT_SME2p1);
```

### Library pseudocode for aarch64/functions/sme/HaveSMEB16B16

```
// HaveSMEB16B16()
// =====
// Returns TRUE if the SME2.1 non-widening BFloat16 instructions are implemented, FALSE otherwise.

boolean HaveSMEB16B16()
    return IsFeatureImplemented(FEAT_SVE_B16B16);
```

### Library pseudocode for aarch64/functions/sme/HaveSMEF16F16

```
// HaveSMEF16F16()
// =====
// Returns TRUE if the SME2.1 half-precision instructions are implemented, FALSE otherwise.

boolean HaveSMEF16F16()
    return IsFeatureImplemented(FEAT_SME_F16F16);
```

### Library pseudocode for aarch64/functions/sme/HaveSMEF64F64

```
// HaveSMEF64F64()
// =====
// Returns TRUE if the SMEF64F64 extension is implemented, FALSE otherwise.

boolean HaveSMEF64F64()
    return IsFeatureImplemented(FEAT_SME_F64F64);
```

### Library pseudocode for aarch64/functions/sme/HaveSMEI16I64

```
// HaveSMEI16I64()
// =====
// Returns TRUE if the SMEI16I64 extension is implemented, FALSE otherwise.

boolean HaveSMEI16I64()
    return IsFeatureImplemented(FEAT_SME_I16I64);
```

### Library pseudocode for aarch64/functions/sme/Lookup

```
bits(512) _ZT0;
```

### Library pseudocode for aarch64/functions/sme/PredCountTest

```
// PredCountTest()
// =====

bits(4) PredCountTest(integer elements, integer count, boolean invert)
    bit n, z, c, v;
    z = (if count == 0 then '1' else '0');           // none active
    if !invert then
        n = (if count != 0 then '1' else '0');       // first active
        c = (if count == elements then '0' else '1'); // NOT last active
    else
        n = (if count == elements then '1' else '0'); // first active
        c = (if count != 0 then '0' else '1');       // NOT last active
    v = '0';

    return n:z:c:v;
```

### Library pseudocode for aarch64/functions/sme/System

```
// System Registers
// =====

array bits(MAX\_VL) _ZA[0..255];
```

## Library pseudocode for aarch64/functions/sme/ZAhslice

```
// ZAhslice[] - non-assignment form
// =====

bits(width) ZAhslice[integer tile, integer esize, integer slice, integer width]
    assert esize IN {8, 16, 32, 64, 128};
    integer tiles = esize DIV 8;
    assert tile >= 0 && tile < tiles;
    integer slices = CurrentSVL DIV esize;
    assert slice >= 0 && slice < slices;

    return ZAvector[tile + slice * tiles, width];

// ZAhslice[] - assignment form
// =====

ZAhslice[integer tile, integer esize, integer slice, integer width] = bits(width) value
    assert esize IN {8, 16, 32, 64, 128};
    integer tiles = esize DIV 8;
    assert tile >= 0 && tile < tiles;
    integer slices = CurrentSVL DIV esize;
    assert slice >= 0 && slice < slices;

    ZAvector[tile + slice * tiles, width] = value;
```

## Library pseudocode for aarch64/functions/sme/ZAslice

```
// ZAslice[] - non-assignment form
// =====

bits(width) ZAslice[integer tile, integer esize, boolean vertical, integer slice, integer width]
    bits(width) result;

    if vertical then
        result = ZAvslice[tile, esize, slice, width];
    else
        result = ZAhslice[tile, esize, slice, width];

    return result;

// ZAslice[] - assignment form
// =====

ZAslice[integer tile, integer esize, boolean vertical,
    integer slice, integer width] = bits(width) value
    if vertical then
        ZAvslice[tile, esize, slice, width] = value;
    else
        ZAhslice[tile, esize, slice, width] = value;
```

## Library pseudocode for aarch64/functions/sme/ZAtile

```
// ZAtile[] - non-assignment form
// =====

bits(width) ZAtile[integer tile, integer esize, integer width]
    constant integer SVL = CurrentSVL;
    integer slices = SVL DIV esize;
    assert width == SVL * slices;
    bits(width) result;

    for slice = 0 to slices-1
        Elem[result, slice, SVL] = ZAhslice[tile, esize, slice, SVL];

    return result;

// ZAtile[] - assignment form
// =====

ZAtile[integer tile, integer esize, integer width] = bits(width) value
    constant integer SVL = CurrentSVL;
    integer slices = SVL DIV esize;
    assert width == SVL * slices;

    for slice = 0 to slices-1
        ZAhslice[tile, esize, slice, SVL] = Elem[value, slice, SVL];
```

## Library pseudocode for aarch64/functions/sme/ZAvector

```
// ZAvector[] - non-assignment form
// =====

bits(width) ZAvector[integer index, integer width]
    assert width == CurrentSVL;
    assert index >= 0 && index < (width DIV 8);

    return \_ZA[index]<width-1:0>;

// ZAvector[] - assignment form
// =====

ZAvector[integer index, integer width] = bits(width) value
    assert width == CurrentSVL;
    assert index >= 0 && index < (width DIV 8);

    if ConstrainUnpredictableBool(Unpredictable\_SMEZEROUPPER) then
        \_ZA[index] = ZeroExtend(value, MAX\_VL);
    else
        \_ZA[index]<width-1:0> = value;
```

## Library pseudocode for aarch64/functions/sme/ZAvslice

```
// ZAvslice[] - non-assignment form
// =====

bits(width) ZAvslice[integer tile, integer esize, integer slice, integer width]
    integer slices = CurrentSVL DIV esize;
    bits(width) result;

    for s = 0 to slices-1
        bits(width) hslice = ZAhslice[tile, esize, s, width];
        Elem[result, s, esize] = Elem[hslice, slice, esize];

    return result;

// ZAvslice[] - assignment form
// =====

ZAvslice[integer tile, integer esize, integer slice, integer width] = bits(width) value
    integer slices = CurrentSVL DIV esize;

    for s = 0 to slices-1
        bits(width) hslice = ZAhslice[tile, esize, s, width];
        Elem[hslice, slice, esize] = Elem[value, s, esize];
        ZAhslice[tile, esize, s, width] = hslice;
```

## Library pseudocode for aarch64/functions/sme/ZT0

```
// ZT0[] - non-assignment form
// =====

bits(width) ZT0[integer width]
    assert width == 512;
    return _ZT0<width-1:0>;

// ZT0[] - assignment form
// =====

ZT0[integer width] = bits(width) value
    assert width == 512;
    _ZT0<width-1:0> = value;
```

## Library pseudocode for aarch64/functions/sve/AArch32.IsFPEnabled

```
// AArch32.IsFPEnabled()
// =====
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch32 state and FALSE otherwise.

boolean AArch32.IsFPEnabled(bits(2) el)
    if el == EL0 && !ELUsingAArch32(EL1) then
        return AArch64.IsFPEnabled(el);

    if HaveEL(EL3) && ELUsingAArch32(EL3) && CurrentSecurityState() == SS\_NonSecure then
        // Check if access disabled in NSACR
        if NSACR.cp10 == '0' then return FALSE;

    if el IN {EL0, EL1} then
        // Check if access disabled in CPACR
        boolean disabled;
        case CPACR.cp10 of
            when '00' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '10' disabled = ConstrainUnpredictableBool(Unpredictable\_RESCPACR);
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if !ELUsingAArch32(EL2) then
            return AArch64.IsFPEnabled(EL2);
        if HCPTR.TCP10 == '1' then return FALSE;

    if HaveEL(EL3) && !ELUsingAArch32(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then return FALSE;

    return TRUE;
```

## Library pseudocode for aarch64/functions/sve/AArch64.IsFPEnabled

```
// AArch64.IsFPEnabled()
// =====
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch64 state and FALSE otherwise.

boolean AArch64.IsFPEnabled(bits(2) el)
    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SIMD&FP at EL0/EL1
        boolean disabled;
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            boolean disabled;
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TFP == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.TFP == '1' then return FALSE;

    return TRUE;
```

## Library pseudocode for aarch64/functions/sve/ActivePredicateElement

```
// ActivePredicateElement()
// =====
// Returns TRUE if the predicate bit is 1 and FALSE otherwise

boolean ActivePredicateElement(bits(N) pred, integer e, integer esize)
    assert esize IN {8, 16, 32, 64, 128};
    integer n = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred<n> == '1';
```

## Library pseudocode for aarch64/functions/sve/AnyActiveElement

```
// AnyActiveElement()
// =====
// Return TRUE if there is at least one active element in mask. Otherwise,
// return FALSE.

boolean AnyActiveElement(bits(N) mask, integer esize)
    return LastActiveElement(mask, esize) >= 0;
```



### Library pseudocode for aarch64/functions/sve/BitDeposit

```
// BitDeposit()
// =====
// Deposit the least significant bits from DATA into result positions
// selected by nonzero bits in MASK, setting other result bits to zero.

bits(N) BitDeposit (bits(N) data, bits(N) mask)
    bits(N) res = Zeros(N);
    integer db = 0;
    for rb = 0 to N-1
        if mask<rb> == '1' then
            res<rb> = data<db>;
            db = db + 1;
    return res;
```

### Library pseudocode for aarch64/functions/sve/BitExtract

```
// BitExtract()
// =====
// Extract and pack DATA bits selected by the nonzero bits in MASK into
// the least significant result bits, setting other result bits to zero.

bits(N) BitExtract (bits(N) data, bits(N) mask)
    bits(N) res = Zeros(N);
    integer rb = 0;
    for db = 0 to N-1
        if mask<db> == '1' then
            res<rb> = data<db>;
            rb = rb + 1;
    return res;
```

### Library pseudocode for aarch64/functions/sve/BitGroup

```
// BitGroup()
// =====
// Extract and pack DATA bits selected by the nonzero bits in MASK into
// the least significant result bits, and pack unselected bits into the
// most significant result bits.

bits(N) BitGroup (bits(N) data, bits(N) mask)
    bits(N) res;
    integer rb = 0;

    // compress masked bits to right
    for db = 0 to N-1
        if mask<db> == '1' then
            res<rb> = data<db>;
            rb = rb + 1;
    // compress unmasked bits to left
    for db = 0 to N-1
        if mask<db> == '0' then
            res<rb> = data<db>;
            rb = rb + 1;
    return res;
```

### Library pseudocode for aarch64/functions/sve/CeilPow2

```
// CeilPow2()
// =====
// For a positive integer X, return the smallest power of 2 >= X

integer CeilPow2(integer x)
    if x == 0 then return 0;
    if x == 1 then return 2;
    return FloorPow2(x - 1) * 2;
```

## Library pseudocode for aarch64/functions/sve/CheckNonStreamingSVEEnabled

```
// CheckNonStreamingSVEEnabled()
// =====
// Checks for traps on SVE instructions that are not legal in streaming mode.

CheckNonStreamingSVEEnabled()
    CheckSVEEnabled();

    if HaveSME() && PSTATE.SM == '1' && !IsFullA64Enabled() then
        SMEAccessTrap(SMEExceptionType\_Streaming, PSTATE.EL);
```

## Library pseudocode for aarch64/functions/sve/CheckOriginalSVEEnabled

```
// CheckOriginalSVEEnabled()
// =====
// Checks for traps on SVE instructions and instructions that access SVE System
// registers.

CheckOriginalSVEEnabled()
    assert HaveSVE();
    boolean disabled;

    if (HaveEL(EL3) && (CPTR_EL3.EZ == '0' || CPTR_EL3.TFP == '1') &&
        EL3SDDUndefPriority()) then
        UNDEFINED;

    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SVEAccessTrap(EL1);

        // Check SIMD&FP at EL0/EL1
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    // Check if access disabled in CPTR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            // Check SVE at EL2
            case CPTR_EL2.ZEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then SVEAccessTrap(EL2);

            // Check SIMD&FP at EL2
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TZ == '1' then SVEAccessTrap(EL2);
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                SVEAccessTrap(EL3);

        if CPTR_EL3.TFP == '1' then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                AArch64.AdvSIMDFPAccessTrap(EL3);
```

## Library pseudocode for aarch64/functions/sve/CheckSMEAccess

```
// CheckSMEAccess()
// =====
// Check that access to SME System registers is enabled.

CheckSMEAccess()
    boolean disabled;
    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1);

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            // Check SME at EL2
            case CPTR_EL2.SMEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);
        else
            if CPTR_EL2.TSM == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL3);
```

## Library pseudocode for aarch64/functions/sve/CheckSMEAndZAAEnabled

```
// CheckSMEAndZAAEnabled()
// =====

CheckSMEAndZAAEnabled()
    CheckSMEEnabled();

    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);
```

## Library pseudocode for aarch64/functions/sve/CheckSMEEnabled

```
// CheckSMEEnabled()
// =====

CheckSMEEnabled()
    boolean disabled;
    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1);

        // Check SIMD&FP at EL0/EL1
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            // Check SME at EL2
            case CPTR_EL2.SMEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);

            // Check SIMD&FP at EL2
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TSM == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL3);
        if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);
```

### Library pseudocode for aarch64/functions/sve/CheckSMEZT0Enabled

```
// CheckSMEZT0Enabled()
// =====
// Checks for ZT0 enabled.

CheckSMEZT0Enabled()
    // Check if ZA and ZT0 are inactive in PSTATE
    if PSTATE.ZA == '0' then
        SMEAccessTrap\(SMEExceptionType\_InactiveZA, PSTATE.EL\);

    // Check if EL0/EL1 accesses to ZT0 are disabled in SMCR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost\(\) then
        if SMCR_EL1.EZT0 == '0' then
            SMEAccessTrap\(SMEExceptionType\_InaccessibleZT0, EL1\);

    // Check if EL0/EL1/EL2 accesses to ZT0 are disabled in SMCR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled\(\) then
        if SMCR_EL2.EZT0 == '0' then
            SMEAccessTrap\(SMEExceptionType\_InaccessibleZT0, EL2\);

    // Check if all accesses to ZT0 are disabled in SMCR_EL3
    if HaveEL\(EL3\) then
        if SMCR_EL3.EZT0 == '0' then
            SMEAccessTrap\(SMEExceptionType\_InaccessibleZT0, EL3\);
```

### Library pseudocode for aarch64/functions/sve/CheckSVEEnabled

```
// CheckSVEEnabled()
// =====
// Checks for traps on SVE instructions and instructions that
// access SVE System registers.

CheckSVEEnabled()
    if HaveSME\(\) && PSTATE.SM == '1' then
        CheckSMEEnabled\(\);
    elseif HaveSME\(\) && !HaveSVE\(\) then
        CheckStreamingSVEEnabled\(\);
    else
        CheckOriginalSVEEnabled\(\);
```

### Library pseudocode for aarch64/functions/sve/CheckStreamingSVEAndZAEEnabled

```
// CheckStreamingSVEAndZAEEnabled()
// =====

CheckStreamingSVEAndZAEEnabled()
    CheckStreamingSVEEnabled\(\);

    if PSTATE.ZA == '0' then
        SMEAccessTrap\(SMEExceptionType\_InactiveZA, PSTATE.EL\);
```

### Library pseudocode for aarch64/functions/sve/CheckStreamingSVEEnabled

```
// CheckStreamingSVEEnabled()
// =====

CheckStreamingSVEEnabled()
    CheckSMEEnabled\(\);

    if PSTATE.SM == '0' then
        SMEAccessTrap\(SMEExceptionType\_NotStreaming, PSTATE.EL\);
```

## Library pseudocode for aarch64/functions/sve/CurrentNSVL

```
// CurrentNSVL - non-assignment form
// =====
// Non-Streaming VL

integer CurrentNSVL
    integer vl;

    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost\(\)) then
        vl = UInt(ZCR_EL1.LEN);

    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost\(\)) then
        vl = UInt(ZCR_EL2.LEN);
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        vl = Min(vl, UInt(ZCR_EL2.LEN));

    if PSTATE.EL == EL3 then
        vl = UInt(ZCR_EL3.LEN);
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(ZCR_EL3.LEN));

    vl = (vl + 1) * 128;
    vl = ImplementedSVEVectorLength(vl);

    return vl;
```

## Library pseudocode for aarch64/functions/sve/CurrentSVL

```
// CurrentSVL - non-assignment form
// =====
// Streaming SVL

integer CurrentSVL
    integer vl;

    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost\(\)) then
        vl = UInt(SMCR_EL1.LEN);

    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost\(\)) then
        vl = UInt(SMCR_EL2.LEN);
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        vl = Min(vl, UInt(SMCR_EL2.LEN));

    if PSTATE.EL == EL3 then
        vl = UInt(SMCR_EL3.LEN);
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(SMCR_EL3.LEN));

    vl = (vl + 1) * 128;
    vl = ImplementedSMEVectorLength(vl);

    return vl;
```

## Library pseudocode for aarch64/functions/sve/CurrentVL

```
// CurrentVL - non-assignment form
// =====

integer CurrentVL
    return if HaveSME() && PSTATE.SM == '1' then CurrentSVL else CurrentNSVL;
```

## Library pseudocode for aarch64/functions/sve/DecodePredCount

```
// DecodePredCount()
// =====

integer DecodePredCount(bits(5) bitpattern, integer esize)
    integer elements = CurrentVL DIV esize;
    integer numElem;
    case bitpattern of
        when '00000' numElem = FloorPow2(elements);
        when '00001' numElem = if elements >= 1 then 1 else 0;
        when '00010' numElem = if elements >= 2 then 2 else 0;
        when '00011' numElem = if elements >= 3 then 3 else 0;
        when '00100' numElem = if elements >= 4 then 4 else 0;
        when '00101' numElem = if elements >= 5 then 5 else 0;
        when '00110' numElem = if elements >= 6 then 6 else 0;
        when '00111' numElem = if elements >= 7 then 7 else 0;
        when '01000' numElem = if elements >= 8 then 8 else 0;
        when '01001' numElem = if elements >= 16 then 16 else 0;
        when '01010' numElem = if elements >= 32 then 32 else 0;
        when '01011' numElem = if elements >= 64 then 64 else 0;
        when '01100' numElem = if elements >= 128 then 128 else 0;
        when '01101' numElem = if elements >= 256 then 256 else 0;
        when '11101' numElem = elements - (elements MOD 4);
        when '11110' numElem = elements - (elements MOD 3);
        when '11111' numElem = elements;
        otherwise    numElem = 0;
    return numElem;
```

## Library pseudocode for aarch64/functions/sve/ElemFFR

```
// ElemFFR[] - non-assignment form
// =====

bit ElemFFR[integer e, integer esize]
    return PredicateElement(_FFR, e, esize);

// ElemFFR[] - assignment form
// =====

ElemFFR[integer e, integer esize] = bit value
    integer psize = esize DIV 8;
    integer n = e * psize;
    assert n >= 0 && (n + psize) <= CurrentVL DIV 8;
    _FFR<(n+psize)-1:n> = ZeroExtend(value, psize);
    return;
```

## Library pseudocode for aarch64/functions/sve/FFR

```
// FFR[] - non-assignment form
// =====

bits(width) FFR[integer width]
    assert width == CurrentVL DIV 8;
    return _FFR<width-1:0>;

// FFR[] - assignment form
// =====

FFR[integer width] = bits(width) value
    assert width == CurrentVL DIV 8;
    if ConstrainUnpredictableBool(Unpredictable\_SVEZEROUPPER) then
        _FFR = ZeroExtend(value, MAX\_PL);
    else
        _FFR<width-1:0> = value;
```



## Library pseudocode for aarch64/functions/sve/FPCompareNE

```
// FPCompareNE()
// =====

boolean FPCompareNE(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    boolean result;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    op1_nan = type1 IN {FPTType\_SNaN, FPTType\_QNaN};
    op2_nan = type2 IN {FPTType\_SNaN, FPTType\_QNaN};

    if op1_nan || op2_nan then
        result = TRUE;
        if type1 == FPTType\_SNaN || type2 == FPTType\_SNaN then
            FPProcessException(FPExc\_InvalidOp, fpcr);
        else // All non-NaN cases can be evaluated on the values produced by FPUnpack()
            result = (value1 != value2);
            FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

## Library pseudocode for aarch64/functions/sve/FPCompareUN

```
// FPCompareUN()
// =====

boolean FPCompareUN(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if type1 == FPTType\_SNaN || type2 == FPTType\_SNaN then
        FPProcessException(FPExc\_InvalidOp, fpcr);

    result = type1 IN {FPTType\_SNaN, FPTType\_QNaN} || type2 IN {FPTType\_SNaN, FPTType\_QNaN};
    if !result then
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

## Library pseudocode for aarch64/functions/sve/FPConvertSVE

```
// FPConvertSVE()
// =====

bits(M) FPConvertSVE(bits(N) op, FPCRTType fpcr_in, FPRounding rounding, integer M)
    FPCRTType fpcr = fpcr_in;
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, rounding, M);

// FPConvertSVE()
// =====

bits(M) FPConvertSVE(bits(N) op, FPCRTType fpcr_in, integer M)
    FPCRTType fpcr = fpcr_in;
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, FPRoundingMode(fpcr), M);
```

## Library pseudocode for aarch64/functions/sve/FPExpA

```
// FPExpA()
// =====

bits(N) FPExpA(bits(N) op)
    assert N IN {16,32,64};
    bits(N) result;
    bits(N) coeff;
    integer idx = if N == 16 then UInt(op<4:0>) else UInt(op<5:0>);
    coeff = FPExpCoefficient[idx, N];
    if N == 16 then
        result<15:0> = '0':op<9:5>:coeff<9:0>;
    elsif N == 32 then
        result<31:0> = '0':op<13:6>:coeff<22:0>;
    else // N == 64
        result<63:0> = '0':op<16:6>:coeff<51:0>;

    return result;
```



```

// FPExpCoefficient()
// =====

bits(N) FPExpCoefficient(integer index, integer N)
    assert N IN {16,32,64};
    integer result;

    if N == 16 then
        case index of
            when 0 result = 0x0000;
            when 1 result = 0x0016;
            when 2 result = 0x002d;
            when 3 result = 0x0045;
            when 4 result = 0x005d;
            when 5 result = 0x0075;
            when 6 result = 0x008e;
            when 7 result = 0x00a8;
            when 8 result = 0x00c2;
            when 9 result = 0x00dc;
            when 10 result = 0x00f8;
            when 11 result = 0x0114;
            when 12 result = 0x0130;
            when 13 result = 0x014d;
            when 14 result = 0x016b;
            when 15 result = 0x0189;
            when 16 result = 0x01a8;
            when 17 result = 0x01c8;
            when 18 result = 0x01e8;
            when 19 result = 0x0209;
            when 20 result = 0x022b;
            when 21 result = 0x024e;
            when 22 result = 0x0271;
            when 23 result = 0x0295;
            when 24 result = 0x02ba;
            when 25 result = 0x02e0;
            when 26 result = 0x0306;
            when 27 result = 0x032e;
            when 28 result = 0x0356;
            when 29 result = 0x037f;
            when 30 result = 0x03a9;
            when 31 result = 0x03d4;

        elsif N == 32 then
            case index of
                when 0 result = 0x000000;
                when 1 result = 0x0164d2;
                when 2 result = 0x02cd87;
                when 3 result = 0x043a29;
                when 4 result = 0x05aac3;
                when 5 result = 0x071f62;
                when 6 result = 0x08980f;
                when 7 result = 0x0a14d5;
                when 8 result = 0x0b95c2;
                when 9 result = 0x0d1adf;
                when 10 result = 0x0ea43a;
                when 11 result = 0x1031dc;
                when 12 result = 0x11c3d3;
                when 13 result = 0x135a2b;
                when 14 result = 0x14f4f0;
                when 15 result = 0x16942d;
                when 16 result = 0x1837f0;
                when 17 result = 0x19e046;
                when 18 result = 0x1b8d3a;
                when 19 result = 0x1d3eda;
                when 20 result = 0x1ef532;
                when 21 result = 0x20b051;
                when 22 result = 0x227043;
                when 23 result = 0x243516;
                when 24 result = 0x25fed7;
                when 25 result = 0x27cd94;

```

```

when 26 result = 0x29a15b;
when 27 result = 0x2b7a3a;
when 28 result = 0x2d583f;
when 29 result = 0x2f3b79;
when 30 result = 0x3123f6;
when 31 result = 0x3311c4;
when 32 result = 0x3504f3;
when 33 result = 0x36fd92;
when 34 result = 0x38fbaf;
when 35 result = 0x3aff5b;
when 36 result = 0x3d08a4;
when 37 result = 0x3f179a;
when 38 result = 0x412c4d;
when 39 result = 0x4346cd;
when 40 result = 0x45672a;
when 41 result = 0x478d75;
when 42 result = 0x49b9be;
when 43 result = 0x4bec15;
when 44 result = 0x4e248c;
when 45 result = 0x506334;
when 46 result = 0x52a81e;
when 47 result = 0x54f35b;
when 48 result = 0x5744fd;
when 49 result = 0x599dl6;
when 50 result = 0x5bfbb8;
when 51 result = 0x5e60f5;
when 52 result = 0x60ccdf;
when 53 result = 0x633f89;
when 54 result = 0x65b907;
when 55 result = 0x68396a;
when 56 result = 0x6ac0c7;
when 57 result = 0x6d4f30;
when 58 result = 0x6fe4ba;
when 59 result = 0x728177;
when 60 result = 0x75257d;
when 61 result = 0x77d0df;
when 62 result = 0x7a83b3;
when 63 result = 0x7d3e0c;

else // N == 64
  case index of
    when 0 result = 0x00000000000000;
    when 1 result = 0x02C9A3E778061;
    when 2 result = 0x059B0D3158574;
    when 3 result = 0x0874518759BC8;
    when 4 result = 0x0B5586CF9890F;
    when 5 result = 0x0E3EC32D3D1A2;
    when 6 result = 0x11301D0125B51;
    when 7 result = 0x1429AAEA92DE0;
    when 8 result = 0x172B83C7D517B;
    when 9 result = 0x1A35BEB6FCB75;
    when 10 result = 0x1D4873168B9AA;
    when 11 result = 0x2063B88628CD6;
    when 12 result = 0x2387A6E756238;
    when 13 result = 0x26B4565E27CDD;
    when 14 result = 0x29E9DF51FDEE1;
    when 15 result = 0x2D285A6E4030B;
    when 16 result = 0x306FE0A31B715;
    when 17 result = 0x33C08B26416FF;
    when 18 result = 0x371A7373AA9CB;
    when 19 result = 0x3A7DB34E59FF7;
    when 20 result = 0x3DEA64C123422;
    when 21 result = 0x4160A21F72E2A;
    when 22 result = 0x44E086061892D;
    when 23 result = 0x486A2B5C13CD0;
    when 24 result = 0x4BFDAD5362A27;
    when 25 result = 0x4F9B2769D2CA7;
    when 26 result = 0x5342B569D4F82;
    when 27 result = 0x56F4736B527DA;
    when 28 result = 0x5AB07DD485429;

```

```

when 29 result = 0x5E76F15AD2148;
when 30 result = 0x6247EB03A5585;
when 31 result = 0x6623882552225;
when 32 result = 0x6A09E667F3BCD;
when 33 result = 0x6DFB23C651A2F;
when 34 result = 0x71F75E8EC5F74;
when 35 result = 0x75FEB564267C9;
when 36 result = 0x7A11473EB0187;
when 37 result = 0x7E2F336CF4E62;
when 38 result = 0x82589994CCE13;
when 39 result = 0x868D99B4492ED;
when 40 result = 0x8ACE5422AA0DB;
when 41 result = 0x8F1AE99157736;
when 42 result = 0x93737B0CDC5E5;
when 43 result = 0x97D829FDE4E50;
when 44 result = 0x9C49182A3F090;
when 45 result = 0xA0C667B5DE565;
when 46 result = 0xA5503B23E255D;
when 47 result = 0xA9E6B5579FDBF;
when 48 result = 0xAE89F995AD3AD;
when 49 result = 0xB33A2B84F15FB;
when 50 result = 0xB7F76F2FB5E47;
when 51 result = 0xBCC1E904BC1D2;
when 52 result = 0xC199BDD85529C;
when 53 result = 0xC67F12E57D14B;
when 54 result = 0xCB720DCEF9069;
when 55 result = 0xD072D4A07897C;
when 56 result = 0xD5818DCFBA487;
when 57 result = 0xDA9E603DB3285;
when 58 result = 0xDFC97337B9B5F;
when 59 result = 0xE502EE78B3FF6;
when 60 result = 0xEA4AFA2A490DA;
when 61 result = 0xEFA1BEE615A27;
when 62 result = 0xF50765B6E4540;
when 63 result = 0xFA7C1819E90D8;

```

```
return result<N-1:0>;
```

## Library pseudocode for aarch64/functions/sve/FPLogB

```

// FPLogB()
// =====

bits(N) FPLogB(bits(N) op, FPCRTType fpcr)
  assert N IN {16,32,64};
  integer result;
  (fptype,sign,value) = FPUnpack(op, fpcr);

  if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN || fptype == FPTType\_Zero then
    FPProcessException(FPExc\_InvalidOp, fpcr);
    result = -(2^(N-1)); // MinInt, 100..00
  elsif fptype == FPTType\_Infinity then
    result = 2^(N-1) - 1; // MaxInt, 011..11
  else
    // FPUnpack has already scaled a subnormal input
    value = Abs(value);
    result = 0;
    while value < 1.0 do
      value = value * 2.0;
      result = result - 1;
    while value >= 2.0 do
      value = value / 2.0;
      result = result + 1;

    FPProcessDenorm(fptype, N, fpcr);

  return result<N-1:0>;

```

## Library pseudocode for aarch64/functions/sve/FPMinNormal

```
// FPMinNormal()
// =====

bits(N) FPMinNormal(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = Zeros(E-1):'1';
    frac = Zeros(F);
    return sign : exp : frac;
```

## Library pseudocode for aarch64/functions/sve/FPOne

```
// FPOne()
// =====

bits(N) FPOne(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-1);
    frac = Zeros(F);
    return sign : exp : frac;
```

## Library pseudocode for aarch64/functions/sve/FPPointFive

```
// FPPointFive()
// =====

bits(N) FPPointFive(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-2):'0';
    frac = Zeros(F);
    return sign : exp : frac;
```

## Library pseudocode for aarch64/functions/sve/FPScale

```
// FPScale()
// =====

bits(N) FPScale(bits(N) op, integer scale, FPCRTType fpcr)
    assert N IN {16,32,64};
    bits(N) result;
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype == FPTType\_SNaN || fptype == FPType\_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPTType\_Zero then
        result = FPZero(sign, N);
    elsif fptype == FPTType\_Infinity then
        result = FPInfinity(sign, N);
    else
        result = FPRound(value * (2.0^scale), fpcr, N);
        FPProcessDenorm(fptype, N, fpcr);

    return result;
```

## Library pseudocode for aarch64/functions/sve/FPTrigMAdd

```
// FPTrigMAdd()
// =====

bits(N) FPTrigMAdd(integer x_in, bits(N) op1, bits(N) op2_in, FPCRTType fpcr)
    assert N IN {16,32,64};
    bits(N) coeff;
    bits(N) op2 = op2_in;
    integer x = x_in;
    assert x >= 0;
    assert x < 8;

    if op2<N-1> == '1' then
        x = x + 8;

    coeff    = FPTrigMAddCoefficient[x, N];
    op2      = FPAbs(op2);
    result   = FPMulAdd(coeff, op1, op2, fpcr);
    return result;
```



## Library pseudocode for aarch64/functions/sve/FPTrigMAddCoefficient

```
// FPTrigMAddCoefficient()
// =====

bits(N) FPTrigMAddCoefficient[integer index, integer N]
  assert N IN {16,32,64};
  integer result;

  if N == 16 then
    case index of
      when 0 result = 0x3c00;
      when 1 result = 0xb155;
      when 2 result = 0x2030;
      when 3 result = 0x0000;
      when 4 result = 0x0000;
      when 5 result = 0x0000;
      when 6 result = 0x0000;
      when 7 result = 0x0000;
      when 8 result = 0x3c00;
      when 9 result = 0xb800;
      when 10 result = 0x293a;
      when 11 result = 0x0000;
      when 12 result = 0x0000;
      when 13 result = 0x0000;
      when 14 result = 0x0000;
      when 15 result = 0x0000;
    elseif N == 32 then
      case index of
        when 0 result = 0x3f800000;
        when 1 result = 0xbe2aaaab;
        when 2 result = 0x3c088886;
        when 3 result = 0xb95008b9;
        when 4 result = 0x36369d6d;
        when 5 result = 0x00000000;
        when 6 result = 0x00000000;
        when 7 result = 0x00000000;
        when 8 result = 0x3f800000;
        when 9 result = 0xbf000000;
        when 10 result = 0x3d2aaaa6;
        when 11 result = 0xbab60705;
        when 12 result = 0x37cd37cc;
        when 13 result = 0x00000000;
        when 14 result = 0x00000000;
        when 15 result = 0x00000000;
      else // N == 64
        case index of
          when 0 result = 0x3ff0000000000000;
          when 1 result = 0xbfc5555555555543;
          when 2 result = 0x3f8111111110f30c;
          when 3 result = 0xbf2a01a019b92fc6;
          when 4 result = 0x3ec71de351f3d22b;
          when 5 result = 0xbe5ae5e2b60f7b91;
          when 6 result = 0x3de5d8408868552f;
          when 7 result = 0x0000000000000000;
          when 8 result = 0x3ff0000000000000;
          when 9 result = 0xbfe0000000000000;
          when 10 result = 0x3fa5555555555536;
          when 11 result = 0xbf56c16c16c13a0b;
          when 12 result = 0x3efa01a019b1e8d8;
          when 13 result = 0xbe927e4f7282f468;
          when 14 result = 0x3e21ee96d2641b13;
          when 15 result = 0xbda8f76380fbb401;

  return result<N-1:0>;
```

### Library pseudocode for aarch64/functions/sve/FPTrigSMul

```
// FPTrigSMul()
// =====

bits(N) FPTrigSMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    result = FPMul(op1, op1, fpcr);
    fpexc = FALSE;
    (fptype, sign, value) = FPUnpack(result, fpcr, fpexc);

    if !(fptype IN {FPTType\_QNaN, FPTType\_SNaN}) then
        result<N-1> = op2<0>;

    return result;
```

### Library pseudocode for aarch64/functions/sve/FPTrigSSel

```
// FPTrigSSel()
// =====

bits(N) FPTrigSSel(bits(N) op1, bits(N) op2)
    assert N IN {16,32,64};
    bits(N) result;

    if op2<0> == '1' then
        result = FPOne(op2<1>, N);
    elsif op2<1> == '1' then
        result = FPNeg(op1);
    else
        result = op1;

    return result;
```

### Library pseudocode for aarch64/functions/sve/FirstActive

```
// FirstActive()
// =====

bit FirstActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ActivePredicateElement(mask, e, esize) then
            return PredicateElement(x, e, esize);
    return '0';
```

### Library pseudocode for aarch64/functions/sve/FloorPow2

```
// FloorPow2()
// =====
// For a positive integer X, return the largest power of 2 <= X

integer FloorPow2(integer x)
    assert x >= 0;
    integer n = 1;
    if x == 0 then return 0;
    while x >= 2^n do
        n = n + 1;
    return 2^(n - 1);
```

### Library pseudocode for aarch64/functions/sve/HaveSMEFullA64

```
// HaveSMEFullA64()
// =====
// Returns TRUE if the SME FA64 extension is implemented, FALSE otherwise.

boolean HaveSMEFullA64()
    return IsFeatureImplemented(FEAT_SME_FA64);
```

### Library pseudocode for aarch64/functions/sve/HaveSVE

```
// HaveSVE()
// =====

boolean HaveSVE()
    return IsFeatureImplemented(FEAT_SVE);
```

### Library pseudocode for aarch64/functions/sve/HaveSVE2

```
// HaveSVE2()
// =====
// Returns TRUE if the SVE2 extension is implemented, FALSE otherwise.

boolean HaveSVE2()
    return IsFeatureImplemented(FEAT_SVE2);
```

### Library pseudocode for aarch64/functions/sve/HaveSVE2AES

```
// HaveSVE2AES()
// =====
// Returns TRUE if the SVE2 AES extension is implemented, FALSE otherwise.

boolean HaveSVE2AES()
    return IsFeatureImplemented(FEAT_SVE_AES);
```

### Library pseudocode for aarch64/functions/sve/HaveSVE2BitPerm

```
// HaveSVE2BitPerm()
// =====
// Returns TRUE if the SVE2 Bit Permissions extension is implemented, FALSE otherwise.

boolean HaveSVE2BitPerm()
    return IsFeatureImplemented(FEAT_SVE_BitPerm);
```

### Library pseudocode for aarch64/functions/sve/HaveSVE2PMULL128

```
// HaveSVE2PMULL128()
// =====
// Returns TRUE if the SVE2 128 bit PMULL extension is implemented, FALSE otherwise.

boolean HaveSVE2PMULL128()
    return IsFeatureImplemented(FEAT_SVE_PMULL128);
```

### Library pseudocode for aarch64/functions/sve/HaveSVE2SHA256

```
// HaveSVE2SHA256()
// =====
// Returns TRUE if the SVE2 SHA256 extension is implemented, FALSE otherwise.

boolean HaveSVE2SHA256()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 SHA256 extension";
```

### Library pseudocode for aarch64/functions/sve/HaveSVE2SHA3

```
// HaveSVE2SHA3()
// =====
// Returns TRUE if the SVE2 SHA3 extension is implemented, FALSE otherwise.

boolean HaveSVE2SHA3()
    return IsFeatureImplemented(FEAT_SVE_SHA3);
```

### Library pseudocode for aarch64/functions/sve/HaveSVE2SHA512

```
// HaveSVE2SHA512()
// =====
// Returns TRUE if the SVE2 SHA512 extension is implemented, FALSE otherwise.

boolean HaveSVE2SHA512()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 SHA512 extension";
```

### Library pseudocode for aarch64/functions/sve/HaveSVE2SM3

```
// HaveSVE2SM3()
// =====
// Returns TRUE if the SVE2 SM3 extension is implemented, FALSE otherwise.

boolean HaveSVE2SM3()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 SM3 extension";
```

### Library pseudocode for aarch64/functions/sve/HaveSVE2SM4

```
// HaveSVE2SM4()
// =====
// Returns TRUE if the SVE2 SM4 extension is implemented, FALSE otherwise.

boolean HaveSVE2SM4()
    return IsFeatureImplemented(FEAT_SVE_SM4);
```

### Library pseudocode for aarch64/functions/sve/HaveSVE2p1

```
// HaveSVE2p1()
// =====
// Returns TRUE if the SVE2.1 extension is implemented, FALSE otherwise.

boolean HaveSVE2p1()
    return IsFeatureImplemented(FEAT_SVE2p1);
```

### Library pseudocode for aarch64/functions/sve/HaveSVEB16B16

```
// HaveSVEB16B16()
// =====
// Returns TRUE if the SVE2.1 non-widening BFloat16 instructions are implemented, FALSE otherwise.

boolean HaveSVEB16B16()
    return IsFeatureImplemented(FEAT_SVE_B16B16);
```

### Library pseudocode for aarch64/functions/sve/HaveSVEFP32MatMulExt

```
// HaveSVEFP32MatMulExt()
// =====
// Returns TRUE if single-precision floating-point matrix multiply instruction support implemented
// and FALSE otherwise.

boolean HaveSVEFP32MatMulExt()
    return IsFeatureImplemented(FEAT_F32MM);
```

## Library pseudocode for aarch64/functions/sve/HaveSVEFP64MatMulExt

```
// HaveSVEFP64MatMulExt()
// =====
// Returns TRUE if double-precision floating-point matrix multiply instruction support implemented
// and FALSE otherwise.

boolean HaveSVEFP64MatMulExt()
    return IsFeatureImplemented(FEAT_F64MM);
```

## Library pseudocode for aarch64/functions/sve/ImplementedSMEVectorLength

```
// ImplementedSMEVectorLength()
// =====
// Reduce SVE/SME vector length to a supported value (power of two)

integer ImplementedSMEVectorLength(integer nbits_in)
    integer maxbits = MaxImplementedSVL();
    assert 128 <= maxbits && maxbits <= 2048 && IsPow2(maxbits);
    integer nbits = Min(nbits_in, maxbits);
    assert 128 <= nbits && nbits <= 2048 && Align(nbits, 128) == nbits;

    // Search for a supported power-of-two VL less than or equal to nbits
    while nbits > 128 do
        if IsPow2(nbits) && SupportedPowerTwoSVL(nbits) then return nbits;
        nbits = nbits - 128;

    // Return the smallest supported power-of-two VL
    nbits = 128;
    while nbits < maxbits do
        if SupportedPowerTwoSVL(nbits) then return nbits;
        nbits = nbits * 2;

    // The only option is maxbits
    return maxbits;
```

## Library pseudocode for aarch64/functions/sve/ImplementedSVEVectorLength

```
// ImplementedSVEVectorLength()
// =====
// Reduce SVE vector length to a supported value (power of two)

integer ImplementedSVEVectorLength(integer nbits_in)
    integer maxbits = MaxImplementedVL();
    assert 128 <= maxbits && maxbits <= 2048 && IsPow2(maxbits);
    integer nbits = Min(nbits_in, maxbits);
    assert 128 <= nbits && nbits <= 2048 && Align(nbits, 128) == nbits;

    while nbits > 128 do
        if IsPow2(nbits) then return nbits;
        nbits = nbits - 128;
    return nbits;
```

## Library pseudocode for aarch64/functions/sve/InStreamingMode

```
// InStreamingMode()
// =====

boolean InStreamingMode()
    return HaveSME() && PSTATE.SM == '1';
```

### Library pseudocode for aarch64/functions/sve/IsEven

```
// IsEven()
// =====

boolean IsEven(integer val)
    return val MOD 2 == 0;
```

### Library pseudocode for aarch64/functions/sve/IsFPEnabled

```
// IsFPEnabled()
// =====
// Returns TRUE if accesses to the Advanced SIMD and floating-point
// registers are enabled at the target exception level in the current
// execution state and FALSE otherwise.

boolean IsFPEnabled(bits(2) el)
    if ELUsingAArch32(el) then
        return AArch32.IsFPEnabled(el);
    else
        return AArch64.IsFPEnabled(el);
```

### Library pseudocode for aarch64/functions/sve/IsFullA64Enabled

```
// IsFullA64Enabled()
// =====
// Returns TRUE is full A64 is enabled in Streaming mode and FALSE otherwise.

boolean IsFullA64Enabled()
    if !HaveSMEFullA64() then return FALSE;

    // Check if full SVE disabled in SMCR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check full SVE at EL0/EL1
        if SMCR_EL1.FA64 == '0' then return FALSE;

    // Check if full SVE disabled in SMCR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if SMCR_EL2.FA64 == '0' then return FALSE;

    // Check if full SVE disabled in SMCR_EL3
    if HaveEL(EL3) then
        if SMCR_EL3.FA64 == '0' then return FALSE;

    return TRUE;
```

### Library pseudocode for aarch64/functions/sve/IsOdd

```
// IsOdd()
// =====

boolean IsOdd(integer val)
    return val MOD 2 == 1;
```

## Library pseudocode for aarch64/functions/sve/IsOriginalSVEEnabled

```
// IsOriginalSVEEnabled()
// =====
// Returns TRUE if access to SVE functionality is enabled at the target
// exception level and FALSE otherwise.

boolean IsOriginalSVEEnabled(bits(2) el)
    boolean disabled;
    if ELUsingAArch32(el) then
        return FALSE;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            case CPTR_EL2.ZEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TZ == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then return FALSE;

    return TRUE;
```

## Library pseudocode for aarch64/functions/sve/IsPow2

```
// IsPow2()
// =====
// Return TRUE if positive integer X is a power of 2. Otherwise,
// return FALSE.

boolean IsPow2(integer x)
    if x <= 0 then return FALSE;
    return FloorPow2(x) == CeilPow2(x);
```

## Library pseudocode for aarch64/functions/sve/IsSMEEnabled

```
// IsSMEEnabled()
// =====
// Returns TRUE if access to SME functionality is enabled at the target
// exception level and FALSE otherwise.

boolean IsSMEEnabled(bits(2) el)
    boolean disabled;
    if ELUsingAArch32(el) then
        return FALSE;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            case CPTR_EL2.SMEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TSM == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then return FALSE;

    return TRUE;
```

## Library pseudocode for aarch64/functions/sve/IsSVEEnabled

```
// IsSVEEnabled()
// =====
// Returns TRUE if access to SVE registers is enabled at the target exception
// level and FALSE otherwise.

boolean IsSVEEnabled(bits(2) el)
    if HaveSME() && PSTATE.SM == '1' then
        return IsSMEEnabled(el);
    elseif HaveSVE() then
        return IsOriginalSVEEnabled(el);
    else
        return FALSE;
```

## Library pseudocode for aarch64/functions/sve/LastActive

```
// LastActive()
// =====

bit LastActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = elements-1 downto 0
        if ActivePredicateElement(mask, e, esize) then
            return PredicateElement(x, e, esize);
    return '0';
```



### Library pseudocode for aarch64/functions/sve/LastActiveElement

```
// LastActiveElement()
// =====

integer LastActiveElement(bits(N) mask, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = elements-1 downto 0
        if ActivePredicateElement(mask, e, esize) then return e;
    return -1;
```

### Library pseudocode for aarch64/functions/sve/MaxImplementedSVL

```
// MaxImplementedSVL()
// =====

integer MaxImplementedSVL()
    return integer IMPLEMENTATION_DEFINED "Max implemented SVL";
```

### Library pseudocode for aarch64/functions/sve/MaxImplementedVL

```
// MaxImplementedVL()
// =====

integer MaxImplementedVL()
    return integer IMPLEMENTATION_DEFINED "Max implemented VL";
```

## Library pseudocode for aarch64/functions/sve/MaybeZeroSVEUppers

```
// MaybeZeroSVEUppers()
// =====

MaybeZeroSVEUppers(bits(2) target_el)
    boolean lower_enabled;

    if UInt(target_el) <= UInt(PSTATE.EL) || !IsSVEEnabled(target_el) then
        return;

    if target_el == EL3 then
        if EL2Enabled() then
            lower_enabled = IsFPEEnabled(EL2);
        else
            lower_enabled = IsFPEEnabled(EL1);
    elseif target_el == EL2 then
        assert EL2Enabled() && !ELUsingAArch32(EL2);
        if HCR_EL2.TGE == '0' then
            lower_enabled = IsFPEEnabled(EL1);
        else
            lower_enabled = IsFPEEnabled(EL0);
    else
        assert target_el == EL1 && !ELUsingAArch32(EL1);
        lower_enabled = IsFPEEnabled(EL0);

    if lower_enabled then
        constant integer VL = if IsSVEEnabled(PSTATE.EL) then CurrentVL else 128;
        constant integer PL = VL DIV 8;
        for n = 0 to 31
            if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
                _Z[n] = ZeroExtend(_Z[n]<VL-1:0>, MAX_VL);
        for n = 0 to 15
            if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
                _P[n] = ZeroExtend(_P[n]<PL-1:0>, MAX_PL);
        if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
            _FFR = ZeroExtend(_FFR<PL-1:0>, MAX_PL);
        if HaveSME() && PSTATE.ZA == '1' then
            constant integer SVL = CurrentSVL;
            constant integer accessiblevecs = SVL DIV 8;
            constant integer allvecs = MaxImplementedSVL() DIV 8;

            for n = 0 to accessiblevecs - 1
                if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
                    _ZA[n] = ZeroExtend(_ZA[n]<SVL-1:0>, MAX_VL);
            for n = accessiblevecs to allvecs - 1
                if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
                    _ZA[n] = Zeros(MAX_VL);
```

## Library pseudocode for aarch64/functions/sve/MemNF

```
// MemNF[] - non-assignment form
// =====

(bits(8*size), boolean) MemNF[bits(64) address, integer size, AccessDescriptor accdesc]
    assert size IN {1, 2, 4, 8, 16};
    bits(8*size) value;
    boolean bad;

    boolean aligned = IsAligned(address, size);

    if !aligned && AlignmentEnforced() then
        return (bits(8*size) UNKNOWN, TRUE);

    boolean atomic = aligned || size == 1;

    if !atomic then
        (value<7:0>, bad) = MemSingleNF[address, 1, accdesc, aligned];

        if bad then
            return (bits(8*size) UNKNOWN, TRUE);

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable\_DEVPAGE2);
            assert c IN {Constraint\_FAULT, Constraint\_NONE};
            if c == Constraint\_NONE then aligned = TRUE;

        for i = 1 to size-1
            (value<8*i+7:8*i>, bad) = MemSingleNF[address+i, 1, accdesc, aligned];

            if bad then
                return (bits(8*size) UNKNOWN, TRUE);
    else
        (value, bad) = MemSingleNF[address, size, accdesc, aligned];
        if bad then
            return (bits(8*size) UNKNOWN, TRUE);

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);

    return (value, FALSE);
```

## Library pseudocode for aarch64/functions/sve/MemSingleNF

```
// MemSingleNF[] - non-assignment form
// =====

(bits(8*size), boolean) MemSingleNF(bits(64) address, integer size, AccessDescriptor accdesc_in,
                                     boolean aligned]
    assert accdesc_in.acctype == AccessType\_SVE;
    assert accdesc_in.nonfault || (accdesc_in.firstfault && !accdesc_in.first);

    bits(8*size) value;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus;
    AccessDescriptor accdesc = accdesc_in;
    FaultRecord fault = NoFault(accdesc);

    // Implementation may suppress NF load for any reason
    if ConstrainUnpredictableBool(Unpredictable\_NONFAULT) then
        return (bits(8*size) UNKNOWN, TRUE);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if HaveMTE2Ext() && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Non-fault load from Device memory must not be performed externally
    if memaddrdesc.memattrs.memtype == MemType\_Device then
        return (bits(8*size) UNKNOWN, TRUE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (bits(8*size) UNKNOWN, TRUE);

    if HaveMTE2Ext() && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            return (bits(8*size) UNKNOWN, TRUE);

    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        boolean iswrite = FALSE;
        if IsExternalAbortTakenSynchronously(memstatus, iswrite, memaddrdesc, size, accdesc) then
            return (bits(8*size) UNKNOWN, TRUE);
        fault.merrorstate = memstatus.merrorstate;
        fault.extflag = memstatus.extflag;
        fault.statuscode = memstatus.statuscode;
        PendSErrorInterrupt(fault);

    return (value, FALSE);
```

## Library pseudocode for aarch64/functions/sve/NoneActive

```
// NoneActive()
// =====

bit NoneActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ActivePredicateElement(mask, e, esize) && ActivePredicateElement(x, e, esize) then
            return '0';
    return '1';
```

## Library pseudocode for aarch64/functions/sve/P

```
// P[] - non-assignment form
// =====

bits(width) P[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width == CurrentVL DIV 8;
    return _P[n]<width-1:0>;

// P[] - assignment form
// =====

P[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == CurrentVL DIV 8;
    if ConstrainUnpredictableBool(Unpredictable\_SVEZERoupper) then
        _P[n] = ZeroExtend(value, MAX\_PL);
    else
        _P[n]<width-1:0> = value;
```

## Library pseudocode for aarch64/functions/sve/PredTest

```
// PredTest()
// =====

bits(4) PredTest(bits(N) mask, bits(N) result, integer esize)
    bit n = FirstActive(mask, result, esize);
    bit z = NoneActive(mask, result, esize);
    bit c = NOT LastActive(mask, result, esize);
    bit v = '0';
    return n:z:c:v;
```

## Library pseudocode for aarch64/functions/sve/PredicateElement

```
// PredicateElement()
// =====
// Returns the predicate bit

bit PredicateElement(bits(N) pred, integer e, integer esize)
    assert esize IN {8, 16, 32, 64, 128};
    integer n = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred<n>;
```

## Library pseudocode for aarch64/functions/sve/ReducePredicated

```
// ReducePredicated()
// =====

bits(esize) ReducePredicated(ReduceOp op, bits(N) input, bits(M) mask, bits(esize) identity)
    assert(N == M * 8);
    integer p2bits = CeilPow2(N);
    bits(p2bits) operand;
    integer elements = p2bits DIV esize;

    for e = 0 to elements-1
        if e * esize < N && ActivePredicateElement(mask, e, esize) then
            Elem[operand, e, esize] = Elem[input, e, esize];
        else
            Elem[operand, e, esize] = identity;

    return Reduce(op, operand, esize);
```

### Library pseudocode for aarch64/functions/sve/ResetSMEState

```
// ResetSMEState()
// =====

ResetSMEState()
    integer vectors = MAX\_VL DIV 8;
    for n = 0 to vectors - 1
        _ZA[n] = Zeros(MAX\_VL);
    _ZT0 = Zeros(ZT0\_LEN);
```

### Library pseudocode for aarch64/functions/sve/ResetSVEState

```
// ResetSVEState()
// =====

ResetSVEState()
    for n = 0 to 31
        _Z[n] = Zeros(MAX\_VL);
    for n = 0 to 15
        _P[n] = Zeros(MAX\_PL);
    _FFR = Zeros(MAX\_PL);
    FPSR = ZeroExtend(0x0800009f<31:0>, 64);
```

### Library pseudocode for aarch64/functions/sve/Reverse

```
// Reverse()
// =====
// Reverse subwords of M bits in an N-bit word

bits(N) Reverse(bits(N) word, integer M)
    bits(N) result;
    integer sw = N DIV M;
    assert N == sw * M;
    for s = 0 to sw-1
        Elem[result, (sw - 1) - s, M] = Elem[word, s, M];
    return result;
```

## Library pseudocode for aarch64/functions/sve/SMEAccessTrap

```
// SMEAccessTrap()
// =====
// Trapped access to SME registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SMEAccessTrap(SMEExceptionType etype, bits(2) target_el_in)
    bits(2) target_el = target_el_in;
    assert UInt(target_el) >= UInt(PSTATE.EL);
    if target_el == EL0 then
        target_el = EL1;
    boolean route_to_el2;
    route_to_el2 = PSTATE.EL == EL0 && target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1';

    except = ExceptionSyndrome(Exception_SMEAccessTrap);
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    case etype of
        when SMEExceptionType_AccessTrap
            except.syndrome<2:0> = '000';
        when SMEExceptionType_Streaming
            except.syndrome<2:0> = '001';
        when SMEExceptionType_NotStreaming
            except.syndrome<2:0> = '010';
        when SMEExceptionType_InactiveZA
            except.syndrome<2:0> = '011';
        when SMEExceptionType_InaccessibleZT0
            except.syndrome<2:0> = '100';

    if route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/functions/sve/SMEExceptionType

```
// SMEExceptionType
// =====
enumeration SMEExceptionType {
    SMEExceptionType_AccessTrap,          // SME functionality trapped or disabled
    SMEExceptionType_Streaming,           // Illegal instruction in Streaming SVE mode
    SMEExceptionType_NotStreaming,        // Illegal instruction not in Streaming SVE mode
    SMEExceptionType_InactiveZA,          // Illegal instruction when ZA is inactive
    SMEExceptionType_InaccessibleZT0,     // Access to ZT0 is disabled
};
```

## Library pseudocode for aarch64/functions/sve/SVEAccessTrap

```
// SVEAccessTrap()
// =====
// Trapped access to SVE registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SVEAccessTrap(bits(2) target_el)
    assert UInt(target_el) >= UInt(PSTATE.EL) && target_el != EL0 && HaveEL(target_el);
    route_to_el2 = target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1';

    except = ExceptionSyndrome(Exception_SVEAccessTrap);
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    if route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/functions/sve/SVEComp

```
// SVEComp
// =====

enumeration SVEComp { Cmp_EQ, Cmp_NE, Cmp_GE, Cmp_GT, Cmp_LT, Cmp_LE, Cmp_UN };
```

## Library pseudocode for aarch64/functions/sve/SVEMoveMaskPreferred

```
// SVEMoveMaskPreferred()
// =====
// Return FALSE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single DUP instruction.
// Used as a condition for the preferred MOV<-DUPM alias.

boolean SVEMoveMaskPreferred(bits(13) imm13)
    bits(64) imm;
    (imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE, 64);

    // Check for 8 bit immediates
    if !IsZero(imm<7:0>) then
        // Check for 'ffffffffffffffxy' or '00000000000000xy'
        if IsZero(imm<63:7>) || IsOnes(imm<63:7>) then
            return FALSE;

        // Check for 'ffffffxyffffffxy' or '000000xy000000xy'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<31:7>)) then
            return FALSE;

        // Check for 'ffxyffxyffxyffxy' or '00xy00xy00xy00xy'
        if (imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> &&
            (IsZero(imm<15:7>) || IsOnes(imm<15:7>))) then
            return FALSE;

        // Check for 'xyxyxyxyxyxyxyxy'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> && (imm<15:8> == imm<7:0>) then
            return FALSE;

    // Check for 16 bit immediates
    else
        // Check for 'ffffffffffffxy00' or '000000000000xy00'
        if IsZero(imm<63:15>) || IsOnes(imm<63:15>) then
            return FALSE;

        // Check for 'ffffxy00ffffxy00' or '0000xy000000xy00'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<31:7>)) then
            return FALSE;

        // Check for 'xy00xy00xy00xy00'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> then
            return FALSE;

    return TRUE;
```

## Library pseudocode for aarch64/functions/sve/SetPSTATE\_SM

```
// SetPSTATE_SM()
// =====

SetPSTATE_SM(bit value)
    if PSTATE.SM != value then
        ResetSVEState();
        PSTATE.SM = value;
```



### Library pseudocode for aarch64/functions/sve/SetPSTATE\_SVCR

```
// SetPSTATE_SVCR
// =====

SetPSTATE_SVCR(bits(32) svcr)
  SetPSTATE\_SM(svcr<0>);
  SetPSTATE\_ZA(svcr<1>);
```

### Library pseudocode for aarch64/functions/sve/SetPSTATE\_ZA

```
// SetPSTATE_ZA()
// =====

SetPSTATE_ZA(bit value)
  if PSTATE.ZA != value then
    ResetSMEState();
    PSTATE.ZA = value;
```

### Library pseudocode for aarch64/functions/sve/ShiftSat

```
// ShiftSat()
// =====

integer ShiftSat(integer shift, integer esize)
  if shift > esize+1 then return esize+1;
  elsif shift < -(esize+1) then return -(esize+1);
  return shift;
```

### Library pseudocode for aarch64/functions/sve/SupportedPowerTwoSVL

```
// SupportedPowerTwoSVL()
// =====
// Return an IMPLEMENTATION DEFINED specific value
// returns TRUE if SVL is supported and is a power of two, FALSE otherwise

boolean SupportedPowerTwoSVL(integer nbits);
```

### Library pseudocode for aarch64/functions/sve/System

```
constant integer MAX_VL = 2048;
constant integer MAX_PL = 256;
constant integer ZT0_LEN = 512;
bits(MAX\_PL) _FFR;

array bits(MAX\_VL) _Z[0..31];

array bits(MAX\_PL) _P[0..15];
```

## Library pseudocode for aarch64/functions/sve/Z

```
// Z[] - non-assignment form
// =====

bits(width) Z[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width == CurrentVL;
    return _Z[n]<width-1:0>;

// Z[] - assignment form
// =====

Z[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == CurrentVL;
    if ConstrainUnpredictableBool(Unpredictable\_SVEZERoupper) then
        _Z[n] = ZeroExtend(value, MAX\_VL);
    else
        _Z[n]<width-1:0> = value;
```

## Library pseudocode for aarch64/functions/syshintop/SystemHintOp

```
// SystemHintOp
// =====
// System Hint instruction types.

enumeration SystemHintOp {
    SystemHintOp_NOP,
    SystemHintOp_YIELD,
    SystemHintOp_WFE,
    SystemHintOp_WFI,
    SystemHintOp_SEV,
    SystemHintOp_SEVL,
    SystemHintOp_DGH,
    SystemHintOp_ESB,
    SystemHintOp_PSB,
    SystemHintOp_TSB,
    SystemHintOp_BTI,
    SystemHintOp_WFET,
    SystemHintOp_WFIT,
    SystemHintOp_CLRBHB,
    SystemHintOp_GCSB,
    SystemHintOp_CHKFEAT,
    SystemHintOp_CSDB
};
```



```

// SysOp()
// =====

SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
  case op1:CRn:CRm:op2 of
    when '000 0111 1000 000' return Sys AT;      // S1E1R
    when '000 0111 1000 001' return Sys AT;      // S1E1W
    when '000 0111 1000 010' return Sys AT;      // S1E0R
    when '000 0111 1000 011' return Sys AT;      // S1E0W
    when '000 0111 1001 000' return Sys AT;      // S1E1RP
    when '000 0111 1001 001' return Sys AT;      // S1E1WP
    when '100 0111 1000 000' return Sys AT;      // S1E2R
    when '100 0111 1000 001' return Sys AT;      // S1E2W
    when '100 0111 1000 100' return Sys AT;      // S12E1R
    when '100 0111 1000 101' return Sys AT;      // S12E1W
    when '100 0111 1000 110' return Sys AT;      // S12E0R
    when '100 0111 1000 111' return Sys AT;      // S12E0W
    when '110 0111 1000 000' return Sys AT;      // S1E3R
    when '110 0111 1000 001' return Sys AT;      // S1E3W
    when '001 0111 0010 100' return Sys BRB;     // IALL
    when '001 0111 0010 101' return Sys BRB;     // INJ
    when '000 0111 0110 001' return Sys DC;      // IVAC
    when '000 0111 0110 010' return Sys DC;      // ISW
    when '000 0111 0110 011' return Sys DC;      // IGVAC
    when '000 0111 0110 100' return Sys DC;      // IGSW
    when '000 0111 0110 101' return Sys DC;      // IGDVAC
    when '000 0111 0110 110' return Sys DC;      // IGDSW
    when '000 0111 1010 010' return Sys DC;      // CSW
    when '000 0111 1010 100' return Sys DC;      // CGSW
    when '000 0111 1010 110' return Sys DC;      // CGDSW
    when '000 0111 1110 010' return Sys DC;      // CISW
    when '000 0111 1110 100' return Sys DC;      // CIGSW
    when '000 0111 1110 110' return Sys DC;      // CIGDSW
    when '011 0111 0100 001' return Sys DC;      // ZVA
    when '011 0111 0100 011' return Sys DC;      // GVA
    when '011 0111 0100 100' return Sys DC;      // GZVA
    when '011 0111 1010 001' return Sys DC;      // CVAC
    when '011 0111 1010 011' return Sys DC;      // CGVAC
    when '011 0111 1010 101' return Sys DC;      // CGDVAC
    when '011 0111 1011 001' return Sys DC;      // CVAU
    when '011 0111 1100 001' return Sys DC;      // CVAP
    when '011 0111 1100 011' return Sys DC;      // CGVAP
    when '011 0111 1100 101' return Sys DC;      // CGDVAP
    when '011 0111 1101 001' return Sys DC;      // CVADP
    when '011 0111 1101 011' return Sys DC;      // CGVADP
    when '011 0111 1101 101' return Sys DC;      // CGDVADP
    when '011 0111 1110 001' return Sys DC;      // CIVAC
    when '011 0111 1110 011' return Sys DC;      // CIGVAC
    when '011 0111 1110 101' return Sys DC;      // CIGDVAC
    when '100 0111 1110 000' return Sys DC;      // CIPAE
    when '100 0111 1110 111' return Sys DC;      // CIGDPAE
    when '110 0111 1110 001' return Sys DC;      // CIPAPA
    when '110 0111 1110 101' return Sys DC;      // CIGDPAPA
    when '000 0111 0001 000' return Sys IC;      // IALLUIS
    when '000 0111 0101 000' return Sys IC;      // IALLU
    when '011 0111 0101 001' return Sys IC;      // IVAU
    when '000 1000 0001 000' return Sys TLBI;    // VMALLE1OS
    when '000 1000 0001 001' return Sys TLBI;    // VAE1OS
    when '000 1000 0001 010' return Sys TLBI;    // ASIDE1OS
    when '000 1000 0001 011' return Sys TLBI;    // VAAE1OS
    when '000 1000 0001 101' return Sys TLBI;    // VALE1OS
    when '000 1000 0001 111' return Sys TLBI;    // VAALE1OS
    when '000 1000 0010 001' return Sys TLBI;    // RVAE1IS
    when '000 1000 0010 011' return Sys TLBI;    // RVAAE1IS
    when '000 1000 0010 101' return Sys TLBI;    // RVALE1IS
    when '000 1000 0010 111' return Sys TLBI;    // RVAALE1IS
    when '000 1000 0011 000' return Sys TLBI;    // VMALLE1IS
    when '000 1000 0011 001' return Sys TLBI;    // VAE1IS
    when '000 1000 0011 010' return Sys TLBI;    // ASIDE1IS
    when '000 1000 0011 011' return Sys TLBI;    // VAAE1IS

```

```

when '000 1000 0011 101' return Sys TLBI; // VALE1IS
when '000 1000 0011 111' return Sys TLBI; // VAALE1IS
when '000 1000 0101 001' return Sys TLBI; // RVAE1OS
when '000 1000 0101 011' return Sys TLBI; // RVAAE1OS
when '000 1000 0101 101' return Sys TLBI; // RVALE1OS
when '000 1000 0101 111' return Sys TLBI; // RVAALE1OS
when '000 1000 0110 001' return Sys TLBI; // RVAE1
when '000 1000 0110 011' return Sys TLBI; // RVAAE1
when '000 1000 0110 101' return Sys TLBI; // RVALE1
when '000 1000 0110 111' return Sys TLBI; // RVAALE1
when '000 1000 0111 000' return Sys TLBI; // VMALLE1
when '000 1000 0111 001' return Sys TLBI; // VAE1
when '000 1000 0111 010' return Sys TLBI; // ASIDE1
when '000 1000 0111 011' return Sys TLBI; // VAAE1
when '000 1000 0111 101' return Sys TLBI; // VALE1
when '000 1000 0111 111' return Sys TLBI; // VAALE1
when '000 1001 0001 000' return Sys TLBI; // VMALLE1OSNXS
when '000 1001 0001 001' return Sys TLBI; // VAE1OSNXS
when '000 1001 0001 010' return Sys TLBI; // ASIDE1OSNXS
when '000 1001 0001 011' return Sys TLBI; // VAAE1OSNXS
when '000 1001 0001 101' return Sys TLBI; // VALE1OSNXS
when '000 1001 0001 111' return Sys TLBI; // VAALE1OSNXS
when '000 1001 0010 001' return Sys TLBI; // RVAE1ISNXS
when '000 1001 0010 011' return Sys TLBI; // RVAAE1ISNXS
when '000 1001 0010 101' return Sys TLBI; // RVALE1ISNXS
when '000 1001 0010 111' return Sys TLBI; // RVAALE1ISNXS
when '000 1001 0011 000' return Sys TLBI; // VMALLE1ISNXS
when '000 1001 0011 001' return Sys TLBI; // VAE1ISNXS
when '000 1001 0011 010' return Sys TLBI; // ASIDE1ISNXS
when '000 1001 0011 011' return Sys TLBI; // VAAE1ISNXS
when '000 1001 0011 101' return Sys TLBI; // VALE1ISNXS
when '000 1001 0011 111' return Sys TLBI; // VAALE1ISNXS
when '000 1001 0101 001' return Sys TLBI; // RVAE1OSNXS
when '000 1001 0101 011' return Sys TLBI; // RVAAE1OSNXS
when '000 1001 0101 101' return Sys TLBI; // RVALE1OSNXS
when '000 1001 0101 111' return Sys TLBI; // RVAALE1OSNXS
when '000 1001 0110 001' return Sys TLBI; // RVAE1NXS
when '000 1001 0110 011' return Sys TLBI; // RVAAE1NXS
when '000 1001 0110 101' return Sys TLBI; // RVALE1NXS
when '000 1001 0110 111' return Sys TLBI; // RVAALE1NXS
when '000 1001 0111 000' return Sys TLBI; // VMALLE1NXS
when '000 1001 0111 001' return Sys TLBI; // VAE1NXS
when '000 1001 0111 010' return Sys TLBI; // ASIDE1NXS
when '000 1001 0111 011' return Sys TLBI; // VAAE1NXS
when '000 1001 0111 101' return Sys TLBI; // VALE1NXS
when '000 1001 0111 111' return Sys TLBI; // VAALE1NXS
when '100 1000 0000 001' return Sys TLBI; // IPAS2E1IS
when '100 1000 0000 010' return Sys TLBI; // RIPAS2E1IS
when '100 1000 0000 101' return Sys TLBI; // IPAS2LE1IS
when '100 1000 0000 110' return Sys TLBI; // RIPAS2LE1IS
when '100 1000 0001 000' return Sys TLBI; // ALLE2OS
when '100 1000 0001 001' return Sys TLBI; // VAE2OS
when '100 1000 0001 100' return Sys TLBI; // ALLE1OS
when '100 1000 0001 101' return Sys TLBI; // VALE2OS
when '100 1000 0001 110' return Sys TLBI; // VMALLS12E1OS
when '100 1000 0010 001' return Sys TLBI; // RVAE2IS
when '100 1000 0010 101' return Sys TLBI; // RVALE2IS
when '100 1000 0011 000' return Sys TLBI; // ALLE2IS
when '100 1000 0011 001' return Sys TLBI; // VAE2IS
when '100 1000 0011 100' return Sys TLBI; // ALLE1IS
when '100 1000 0011 101' return Sys TLBI; // VALE2IS
when '100 1000 0011 110' return Sys TLBI; // VMALLS12E1IS
when '100 1000 0100 000' return Sys TLBI; // IPAS2E1OS
when '100 1000 0100 001' return Sys TLBI; // IPAS2E1
when '100 1000 0100 010' return Sys TLBI; // RIPAS2E1
when '100 1000 0100 011' return Sys TLBI; // RIPAS2E1OS
when '100 1000 0100 100' return Sys TLBI; // IPAS2LE1OS
when '100 1000 0100 101' return Sys TLBI; // IPAS2LE1
when '100 1000 0100 110' return Sys TLBI; // RIPAS2LE1
when '100 1000 0100 111' return Sys TLBI; // RIPAS2LE1OS

```

```

when '100 1000 0101 001' return Sys TLBI; // RVAE2OS
when '100 1000 0101 101' return Sys TLBI; // RVALE2OS
when '100 1000 0110 001' return Sys TLBI; // RVAE2
when '100 1000 0110 101' return Sys TLBI; // RVALE2
when '100 1000 0111 000' return Sys TLBI; // ALLE2
when '100 1000 0111 001' return Sys TLBI; // VAE2
when '100 1000 0111 100' return Sys TLBI; // ALLE1
when '100 1000 0111 101' return Sys TLBI; // VALE2
when '100 1000 0111 110' return Sys TLBI; // VMALLS12E1
when '100 1001 0000 001' return Sys TLBI; // IPAS2E1ISNXS
when '100 1001 0000 010' return Sys TLBI; // RIPAS2E1ISNXS
when '100 1001 0000 101' return Sys TLBI; // IPAS2LE1ISNXS
when '100 1001 0000 110' return Sys TLBI; // RIPAS2LE1ISNXS
when '100 1001 0001 000' return Sys TLBI; // ALLE2OSNXS
when '100 1001 0001 001' return Sys TLBI; // VAE2OSNXS
when '100 1001 0001 100' return Sys TLBI; // ALLE1OSNXS
when '100 1001 0001 101' return Sys TLBI; // VALE2OSNXS
when '100 1001 0001 110' return Sys TLBI; // VMALLS12E1OSNXS
when '100 1001 0010 001' return Sys TLBI; // RVAE2ISNXS
when '100 1001 0010 101' return Sys TLBI; // RVALE2ISNXS
when '100 1001 0011 000' return Sys TLBI; // ALLE2ISNXS
when '100 1001 0011 001' return Sys TLBI; // VAE2ISNXS
when '100 1001 0011 100' return Sys TLBI; // ALLE1ISNXS
when '100 1001 0011 101' return Sys TLBI; // VALE2ISNXS
when '100 1001 0011 110' return Sys TLBI; // VMALLS12E1ISNXS
when '100 1001 0100 000' return Sys TLBI; // IPAS2E1OSNXS
when '100 1001 0100 001' return Sys TLBI; // IPAS2E1NXS
when '100 1001 0100 010' return Sys TLBI; // RIPAS2E1NXS
when '100 1001 0100 011' return Sys TLBI; // RIPAS2E1OSNXS
when '100 1001 0100 100' return Sys TLBI; // IPAS2LE1OSNXS
when '100 1001 0100 101' return Sys TLBI; // IPAS2LE1NXS
when '100 1001 0100 110' return Sys TLBI; // RIPAS2LE1NXS
when '100 1001 0100 111' return Sys TLBI; // RIPAS2LE1OSNXS
when '100 1001 0101 001' return Sys TLBI; // RVAE2OSNXS
when '100 1001 0101 101' return Sys TLBI; // RVALE2OSNXS
when '100 1001 0110 001' return Sys TLBI; // RVAE2NXS
when '100 1001 0110 101' return Sys TLBI; // RVALE2NXS
when '100 1001 0111 000' return Sys TLBI; // ALLE2NXS
when '100 1001 0111 001' return Sys TLBI; // VAE2NXS
when '100 1001 0111 100' return Sys TLBI; // ALLE1NXS
when '100 1001 0111 101' return Sys TLBI; // VALE2NXS
when '100 1001 0111 110' return Sys TLBI; // VMALLS12E1NXS
when '110 1000 0001 000' return Sys TLBI; // ALLE3OS
when '110 1000 0001 001' return Sys TLBI; // VAE3OS
when '110 1000 0001 100' return Sys TLBI; // PAALLOS
when '110 1000 0001 101' return Sys TLBI; // VALE3OS
when '110 1000 0010 001' return Sys TLBI; // RVAE3IS
when '110 1000 0010 101' return Sys TLBI; // RVALE3IS
when '110 1000 0011 000' return Sys TLBI; // ALLE3IS
when '110 1000 0011 001' return Sys TLBI; // VAE3IS
when '110 1000 0011 101' return Sys TLBI; // VALE3IS
when '110 1000 0100 011' return Sys TLBI; // RPAOS
when '110 1000 0100 111' return Sys TLBI; // RPALOS
when '110 1000 0101 001' return Sys TLBI; // RVAE3OS
when '110 1000 0101 101' return Sys TLBI; // RVALE3OS
when '110 1000 0110 001' return Sys TLBI; // RVAE3
when '110 1000 0110 101' return Sys TLBI; // RVALE3
when '110 1000 0111 000' return Sys TLBI; // ALLE3
when '110 1000 0111 001' return Sys TLBI; // VAE3
when '110 1000 0111 100' return Sys TLBI; // PAALL
when '110 1000 0111 101' return Sys TLBI; // VALE3
when '110 1001 0001 000' return Sys TLBI; // ALLE3OSNXS
when '110 1001 0001 001' return Sys TLBI; // VAE3OSNXS
when '110 1001 0001 101' return Sys TLBI; // VALE3OSNXS
when '110 1001 0010 001' return Sys TLBI; // RVAE3ISNXS
when '110 1001 0010 101' return Sys TLBI; // RVALE3ISNXS
when '110 1001 0011 000' return Sys TLBI; // ALLE3ISNXS
when '110 1001 0011 001' return Sys TLBI; // VAE3ISNXS
when '110 1001 0011 101' return Sys TLBI; // VALE3ISNXS
when '110 1001 0101 001' return Sys TLBI; // RVAE3OSNXS

```

```

when '110 1001 0101 101' return Sys\_TLBI; // RVALE3OSNXS
when '110 1001 0110 001' return Sys\_TLBI; // RVAE3NXS
when '110 1001 0110 101' return Sys\_TLBI; // RVALE3NXS
when '110 1001 0111 000' return Sys\_TLBI; // ALLE3NXS
when '110 1001 0111 001' return Sys\_TLBI; // VAE3NXS
when '110 1001 0111 101' return Sys\_TLBI; // VALE3NXS
otherwise                    return Sys\_SYS;

```

### Library pseudocode for aarch64/functions/sysop/SystemOp

```

// SystemOp
// =====
// System instruction types.

enumeration SystemOp {Sys_AT, Sys_BRB, Sys_DC, Sys_IC, Sys_TLBI, Sys_SYS};

```





```

// SysOp128()
// =====

SystemOp128 SysOp128(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
  case op1:CRn:CRm:op2 of
    when '000 1000 0001 001' return Sys_TLBIP; // VAE1OS
    when '000 1000 0001 011' return Sys_TLBIP; // VAAE1OS
    when '000 1000 0001 101' return Sys_TLBIP; // VAE1OS
    when '000 1000 0001 111' return Sys_TLBIP; // VAAE1OS
    when '000 1000 0011 001' return Sys_TLBIP; // VAE1IS
    when '000 1000 0011 011' return Sys_TLBIP; // VAAE1IS
    when '000 1000 0011 101' return Sys_TLBIP; // VAE1IS
    when '000 1000 0011 111' return Sys_TLBIP; // VAAE1IS
    when '000 1000 0111 001' return Sys_TLBIP; // VAE1
    when '000 1000 0111 011' return Sys_TLBIP; // VAAE1
    when '000 1000 0111 101' return Sys_TLBIP; // VAE1
    when '000 1000 0111 111' return Sys_TLBIP; // VAAE1
    when '000 1001 0001 001' return Sys_TLBIP; // VAE1OSNXS
    when '000 1001 0001 011' return Sys_TLBIP; // VAAE1OSNXS
    when '000 1001 0001 101' return Sys_TLBIP; // VAE1OSNXS
    when '000 1001 0001 111' return Sys_TLBIP; // VAAE1OSNXS
    when '000 1001 0011 001' return Sys_TLBIP; // VAE1ISNXS
    when '000 1001 0011 011' return Sys_TLBIP; // VAAE1ISNXS
    when '000 1001 0011 101' return Sys_TLBIP; // VAE1ISNXS
    when '000 1001 0011 111' return Sys_TLBIP; // VAAE1ISNXS
    when '000 1001 0111 001' return Sys_TLBIP; // VAE1NXS
    when '000 1001 0111 011' return Sys_TLBIP; // VAAE1NXS
    when '000 1001 0111 101' return Sys_TLBIP; // VAE1NXS
    when '000 1001 0111 111' return Sys_TLBIP; // VAAE1NXS
    when '100 1000 0001 001' return Sys_TLBIP; // VAE2OS
    when '100 1000 0001 101' return Sys_TLBIP; // VAE2OS
    when '100 1000 0011 001' return Sys_TLBIP; // VAE2IS
    when '100 1000 0011 101' return Sys_TLBIP; // VAE2IS
    when '100 1000 0111 001' return Sys_TLBIP; // VAE2
    when '100 1000 0111 101' return Sys_TLBIP; // VAE2
    when '100 1001 0001 001' return Sys_TLBIP; // VAE2OSNXS
    when '100 1001 0001 101' return Sys_TLBIP; // VAE2OSNXS
    when '100 1001 0011 001' return Sys_TLBIP; // VAE2ISNXS
    when '100 1001 0011 101' return Sys_TLBIP; // VAE2ISNXS
    when '100 1001 0111 001' return Sys_TLBIP; // VAE2NXS
    when '100 1001 0111 101' return Sys_TLBIP; // VAE2NXS
    when '110 1000 0001 001' return Sys_TLBIP; // VAE3OS
    when '110 1000 0001 101' return Sys_TLBIP; // VAE3OS
    when '110 1000 0011 001' return Sys_TLBIP; // VAE3IS
    when '110 1000 0011 101' return Sys_TLBIP; // VAE3IS
    when '110 1000 0111 001' return Sys_TLBIP; // VAE3
    when '110 1000 0111 101' return Sys_TLBIP; // VAE3
    when '110 1001 0001 001' return Sys_TLBIP; // VAE3OSNXS
    when '110 1001 0001 101' return Sys_TLBIP; // VAE3OSNXS
    when '110 1001 0011 001' return Sys_TLBIP; // VAE3ISNXS
    when '110 1001 0011 101' return Sys_TLBIP; // VAE3ISNXS
    when '110 1001 0111 001' return Sys_TLBIP; // VAE3NXS
    when '110 1001 0111 101' return Sys_TLBIP; // VAE3NXS
    when '100 1000 0000 001' return Sys_TLBIP; // IPAS2E1IS
    when '100 1000 0000 101' return Sys_TLBIP; // IPAS2LE1IS
    when '100 1000 0100 000' return Sys_TLBIP; // IPAS2E1OS
    when '100 1000 0100 001' return Sys_TLBIP; // IPAS2E1
    when '100 1000 0100 100' return Sys_TLBIP; // IPAS2LE1OS
    when '100 1000 0100 101' return Sys_TLBIP; // IPAS2LE1
    when '100 1001 0000 001' return Sys_TLBIP; // IPAS2E1ISNXS
    when '100 1001 0000 101' return Sys_TLBIP; // IPAS2LE1ISNXS
    when '100 1001 0100 000' return Sys_TLBIP; // IPAS2E1OSNXS
    when '100 1001 0100 001' return Sys_TLBIP; // IPAS2E1NXS
    when '100 1001 0100 100' return Sys_TLBIP; // IPAS2LE1OSNXS
    when '100 1001 0100 101' return Sys_TLBIP; // IPAS2LE1NXS
    when '000 1000 0010 001' return Sys_TLBIP; // RVAE1IS
    when '000 1000 0010 011' return Sys_TLBIP; // RVAE1IS
    when '000 1000 0010 101' return Sys_TLBIP; // RVAE1IS
    when '000 1000 0010 111' return Sys_TLBIP; // RVAE1IS
    when '000 1000 0101 001' return Sys_TLBIP; // RVAE1OS

```

```

when '000 1000 0101 011' return Sys TLBIP; // RVAAE1OS
when '000 1000 0101 101' return Sys TLBIP; // RVALE1OS
when '000 1000 0101 111' return Sys TLBIP; // RVAALE1OS
when '000 1000 0110 001' return Sys TLBIP; // RVAE1
when '000 1000 0110 011' return Sys TLBIP; // RVAAE1
when '000 1000 0110 101' return Sys TLBIP; // RVALE1
when '000 1000 0110 111' return Sys TLBIP; // RVAALE1
when '000 1001 0010 001' return Sys TLBIP; // RVAE1ISNXS
when '000 1001 0010 011' return Sys TLBIP; // RVAAE1ISNXS
when '000 1001 0010 101' return Sys TLBIP; // RVALE1ISNXS
when '000 1001 0010 111' return Sys TLBIP; // RVAALE1ISNXS
when '000 1001 0101 001' return Sys TLBIP; // RVAE1OSNXS
when '000 1001 0101 011' return Sys TLBIP; // RVAAE1OSNXS
when '000 1001 0101 101' return Sys TLBIP; // RVALE1OSNXS
when '000 1001 0101 111' return Sys TLBIP; // RVAALE1OSNXS
when '000 1001 0110 001' return Sys TLBIP; // RVAE1NXS
when '000 1001 0110 011' return Sys TLBIP; // RVAAE1NXS
when '000 1001 0110 101' return Sys TLBIP; // RVALE1NXS
when '000 1001 0110 111' return Sys TLBIP; // RVAALE1NXS
when '100 1000 0010 001' return Sys TLBIP; // RVAE2IS
when '100 1000 0010 101' return Sys TLBIP; // RVALE2IS
when '100 1000 0101 001' return Sys TLBIP; // RVAE2OS
when '100 1000 0101 101' return Sys TLBIP; // RVALE2OS
when '100 1000 0110 001' return Sys TLBIP; // RVAE2
when '100 1000 0110 101' return Sys TLBIP; // RVALE2
when '100 1001 0010 001' return Sys TLBIP; // RVAE2ISNXS
when '100 1001 0010 101' return Sys TLBIP; // RVALE2ISNXS
when '100 1001 0101 001' return Sys TLBIP; // RVAE2OSNXS
when '100 1001 0101 101' return Sys TLBIP; // RVALE2OSNXS
when '100 1001 0110 001' return Sys TLBIP; // RVAE2NXS
when '100 1001 0110 101' return Sys TLBIP; // RVALE2NXS
when '110 1000 0010 001' return Sys TLBIP; // RVAE3IS
when '110 1000 0010 101' return Sys TLBIP; // RVALE3IS
when '110 1000 0101 001' return Sys TLBIP; // RVAE3OS
when '110 1000 0101 101' return Sys TLBIP; // RVALE3OS
when '110 1000 0110 001' return Sys TLBIP; // RVAE3
when '110 1000 0110 101' return Sys TLBIP; // RVALE3
when '110 1001 0010 001' return Sys TLBIP; // RVAE3ISNXS
when '110 1001 0010 101' return Sys TLBIP; // RVALE3ISNXS
when '110 1001 0101 001' return Sys TLBIP; // RVAE3OSNXS
when '110 1001 0101 101' return Sys TLBIP; // RVALE3OSNXS
when '110 1001 0110 001' return Sys TLBIP; // RVAE3NXS
when '110 1001 0110 101' return Sys TLBIP; // RVALE3NXS
when '100 1000 0000 010' return Sys TLBIP; // RIPAS2E1IS
when '100 1000 0000 110' return Sys TLBIP; // RIPAS2LE1IS
when '100 1000 0100 010' return Sys TLBIP; // RIPAS2E1
when '100 1000 0100 011' return Sys TLBIP; // RIPAS2E1OS
when '100 1000 0100 110' return Sys TLBIP; // RIPAS2LE1
when '100 1000 0100 111' return Sys TLBIP; // RIPAS2LE1OS
when '100 1001 0000 010' return Sys TLBIP; // RIPAS2E1ISNXS
when '100 1001 0000 110' return Sys TLBIP; // RIPAS2LE1ISNXS
when '100 1001 0100 010' return Sys TLBIP; // RIPAS2E1NXS
when '100 1001 0100 011' return Sys TLBIP; // RIPAS2E1OSNXS
when '100 1001 0100 110' return Sys TLBIP; // RIPAS2LE1NXS
when '100 1001 0100 111' return Sys TLBIP; // RIPAS2LE1OSNXS
otherwise return Sys SYSP;

```

## Library pseudocode for aarch64/functions/sysop\_128/SystemOp128

```

// SystemOp128()
// =====
// System instruction types.

enumeration SystemOp128 {Sys_TLBIP, Sys_SYSP};

```

## Library pseudocode for aarch64/functions/sysregisters/ELR\_EL

```
// ELR_EL[] - non-assignment form
// =====

bits(64) ELR_EL[bits(2) el]
  bits(64) r;
  case el of
    when EL1   r = ELR_EL1;
    when EL2   r = ELR_EL2;
    when EL3   r = ELR_EL3;
    otherwise Unreachable();
  return r;

// ELR_EL[] - assignment form
// =====

ELR_EL[bits(2) el] = bits(64) value
  bits(64) r = value;
  case el of
    when EL1   ELR_EL1 = r;
    when EL2   ELR_EL2 = r;
    when EL3   ELR_EL3 = r;
    otherwise Unreachable();
  return;
```

## Library pseudocode for aarch64/functions/sysregisters/ELR\_ELx

```
// ELR_ELx[] - non-assignment form
// =====

bits(64) ELR_ELx[]
  assert PSTATE.EL != EL0;
  return ELR\_EL[PSTATE.EL];

// ELR_ELx[] - assignment form
// =====

ELR_ELx[] = bits(64) value
  assert PSTATE.EL != EL0;
  ELR\_EL[PSTATE.EL] = value;
  return;
```

## Library pseudocode for aarch64/functions/sysregisters/ESRType

```
type ESRType;
```

## Library pseudocode for aarch64/functions/sysregisters/ESR\_EL

```
// ESR_EL[] - non-assignment form
// =====

ESRType ESR_EL[bits(2) regime]
  bits(64) r;
  case regime of
    when EL1   r = ESR_EL1;
    when EL2   r = ESR_EL2;
    when EL3   r = ESR_EL3;
    otherwise Unreachable();
  return r;
```

## Library pseudocode for aarch64/functions/sysregisters/ESR\_ELx

```
// ESR_ELx[] - non-assignment form
// =====

ESRType ESR_ELx[]
    return ESR\_EL\[S1TranslationRegime\(\)\];

// ESR_ELx[] - assignment form
// =====

ESR_ELx[] = ESRType value
    ESR\_EL\[S1TranslationRegime\(\)\] = value;
```

## Library pseudocode for aarch64/functions/sysregisters/ES\_EL

```
// ES_EL[] - assignment form
// =====

ESR_EL[bits(2) regime] = ESRType value
    bits(64) r = value;
    case regime of
        when EL1    ESR_EL1 = r;
        when EL2    ESR_EL2 = r;
        when EL3    ESR_EL3 = r;
        otherwise Unreachable\(\);
    return;
```

## Library pseudocode for aarch64/functions/sysregisters/FAR\_EL

```
// FAR_EL[] - non-assignment form
// =====

bits(64) FAR_EL[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1    r = FAR_EL1;
        when EL2    r = FAR_EL2;
        when EL3    r = FAR_EL3;
        otherwise Unreachable\(\);
    return r;

// FAR_EL[] - assignment form
// =====

FAR_EL[bits(2) regime] = bits(64) value
    bits(64) r = value;
    case regime of
        when EL1    FAR_EL1 = r;
        when EL2    FAR_EL2 = r;
        when EL3    FAR_EL3 = r;
        otherwise Unreachable\(\);
    return;
```

## Library pseudocode for aarch64/functions/sysregisters/FAR\_ELx

```
// FAR_ELx[] - non-assignment form
// =====

bits(64) FAR_ELx[]
    return FAR\_EL\[S1TranslationRegime\(\)\];

// FAR_ELx[] - assignment form
// =====

FAR_ELx[] = bits(64) value
    FAR\_EL\[S1TranslationRegime\(\)\] = value;
    return;
```

## Library pseudocode for aarch64/functions/sysregisters/PFAR\_EL

```
// PFAR_EL[] - non-assignment form
// =====

bits(64) PFAR_EL[bits(2) regime]
  assert (HavePFAR() || (regime == EL3 && HaveRME()));
  bits(64) r;
  case regime of
    when EL1 r = PFAR_EL1;
    when EL2 r = PFAR_EL2;
    when EL3 r = MFAR_EL3;
    otherwise Unreachable();
  return r;

// PFAR_EL[] - assignment form
// =====

PFAR_EL[bits(2) regime] = bits(64) value
  bits(64) r = value;
  assert (HavePFAR() || (HaveRME() && regime == EL3));
  case regime of
    when EL1 PFAR_EL1 = r;
    when EL2 PFAR_EL2 = r;
    when EL3 MFAR_EL3 = r;
    otherwise Unreachable();
  return;
```

## Library pseudocode for aarch64/functions/sysregisters/PFAR\_ELx

```
// PFAR_ELx[] - non-assignment form
// =====

bits(64) PFAR_ELx[]
  return PFAR_EL[S1TranslationRegime()];

// PFAR_ELx[] - assignment form
// =====

PFAR_ELx[] = bits(64) value
  PFAR_EL[S1TranslationRegime()] = value;
  return;
```

## Library pseudocode for aarch64/functions/sysregisters/S1PIRType

```
type S1PIRType;
```

## Library pseudocode for aarch64/functions/sysregisters/S1PORType

```
type S1PORType;
```

## Library pseudocode for aarch64/functions/sysregisters/S2PIRType

```
type S2PIRType;
```

## Library pseudocode for aarch64/functions/sysregisters/S2PORType

```
type S2PORType;
```

## Library pseudocode for aarch64/functions/sysregisters/SCTLRType

```
type SCTLRType;
```

### Library pseudocode for aarch64/functions/sysregisters/SCTLR\_EL

```
// SCTLR_EL[] - non-assignment form
// =====

SCTLRType SCTLR_EL[bits(2) regime]
  bits(64) r;
  case regime of
    when EL1   r = SCTLR_EL1;
    when EL2   r = SCTLR_EL2;
    when EL3   r = SCTLR_EL3;
    otherwise Unreachable();
  return r;
```

### Library pseudocode for aarch64/functions/sysregisters/SCTLR\_ELx

```
// SCTLR_ELx[] - non-assignment form
// =====

SCTLRType SCTLR_ELx[]
  return SCTLR\_EL[S1TranslationRegime()];
```

### Library pseudocode for aarch64/functions/sysregisters/VBAR\_EL

```
// VBAR_EL[] - non-assignment form
// =====

bits(64) VBAR_EL[bits(2) regime]
  bits(64) r;
  case regime of
    when EL1   r = VBAR_EL1;
    when EL2   r = VBAR_EL2;
    when EL3   r = VBAR_EL3;
    otherwise Unreachable();
  return r;
```

### Library pseudocode for aarch64/functions/sysregisters/VBAR\_ELx

```
// VBAR_ELx[] - non-assignment form
// =====

bits(64) VBAR_ELx[]
  return VBAR\_EL[S1TranslationRegime()];
```

### Library pseudocode for aarch64/functions/system/AArch64.AllocationTagAccessIsEnabled

```
// AArch64.AllocationTagAccessIsEnabled()
// =====
// Check whether access to Allocation Tags is enabled.

boolean AArch64.AllocationTagAccessIsEnabled(bits(2) el)
  if SCR_EL3.ATA == '0' && el IN {EL0, EL1, EL2} then
    return FALSE;
  if HCR_EL2.ATA == '0' && el IN {EL0, EL1} && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' then
    return FALSE;

  Regime regime = TranslationRegime(el);
  case regime of
    when Regime\_EL3 return SCTLR_EL3.ATA == '1';
    when Regime\_EL2 return SCTLR_EL2.ATA == '1';
    when Regime\_EL20 return if el == EL0 then SCTLR_EL2.ATA0 == '1' else SCTLR_EL2.ATA == '1';
    when Regime\_EL10 return if el == EL0 then SCTLR_EL1.ATA0 == '1' else SCTLR_EL1.ATA == '1';
    otherwise Unreachable();
```

## Library pseudocode for aarch64/functions/system/AArch64.CheckDAIFAccess

```
// AArch64.CheckDAIFAccess()
// =====
// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted.

AArch64.CheckDAIFAccess(PSTATEField field)
    if PSTATE.EL == ELO && field IN {PSTATEField_DAIFSet, PSTATEField_DAIFClr} then
        if IsInHost() || SCTL_R_EL1.UMA == '0' then
            if EL2Enabled() && HCR_EL2.TGE == '1' then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                AArch64.SystemAccessTrap(EL1, 0x18);
```

## Library pseudocode for aarch64/functions/system/AArch64.CheckSystemAccess

```
// AArch64.CheckSystemAccess()
// =====

AArch64.CheckSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn,
                           bits(4) crm, bits(3) op2, bits(5) rt, bit read)
    if HaveBTIExt() then
        BranchTargetCheck();

    if (HaveTME() && TSTATE.depth > 0 &&
        !CheckTransactionalSystemAccess(op0, op1, crn, crm, op2, read)) then
        FailTransaction(TMFailure_ERR, FALSE);

    return;
```

## Library pseudocode for aarch64/functions/system/AArch64.ChooseNonExcludedTag

```
// AArch64.ChooseNonExcludedTag()
// =====
// Return a tag derived from the start and the offset values, excluding
// any tags in the given mask.

bits(4) AArch64.ChooseNonExcludedTag(bits(4) tag_in, bits(4) offset_in, bits(16) exclude)
    bits(4) tag = tag_in;
    bits(4) offset = offset_in;

    if IsOnes(exclude) then
        return '0000';

    if offset == '0000' then
        while exclude<UInt>(tag) == '1' do
            tag = tag + '0001';

    while offset != '0000' do
        offset = offset - '0001';
        tag = tag + '0001';
        while exclude<UInt>(tag) == '1' do
            tag = tag + '0001';

    return tag;
```

### Library pseudocode for aarch64/functions/system/AArch64.ExecutingBROrBLROrRetInstr

```
// AArch64.ExecutingBROrBLROrRetInstr()
// =====
// Returns TRUE if current instruction is a BR, BLR, RET, B[L]RA[B][Z], or RETA[B].

boolean AArch64.ExecutingBROrBLROrRetInstr()
    if !HaveBTIExt() then return FALSE;

    instr = ThisInstr();
    if instr<31:25> == '1101011' && instr<20:16> == '11111' then
        opc = instr<24:21>;
        return opc != '0101';
    else
        return FALSE;
```

### Library pseudocode for aarch64/functions/system/AArch64.ExecutingBTIInstr

```
// AArch64.ExecutingBTIInstr()
// =====
// Returns TRUE if current instruction is a BTI.

boolean AArch64.ExecutingBTIInstr()
    if !HaveBTIExt() then return FALSE;

    instr = ThisInstr();
    if instr<31:22> == '1101010100' && instr<21:12> == '0000110010' && instr<4:0> == '11111' then
        CRm = instr<11:8>;
        op2 = instr<7:5>;
        return (CRm == '0100' && op2<0> == '0');
    else
        return FALSE;
```

### Library pseudocode for aarch64/functions/system/AArch64.ExecutingERETInstr

```
// AArch64.ExecutingERETInstr()
// =====
// Returns TRUE if current instruction is ERET.

boolean AArch64.ExecutingERETInstr()
    instr = ThisInstr();
    return instr<31:12> == '11010110100111110000';
```

### Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysInstr

```
// AArch64.ImpDefSysInstr()
// =====
// Execute an implementation-defined system instruction with write (source operand).

AArch64.ImpDefSysInstr(integer el, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2, integer t);
```

### Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysInstr128

```
// AArch64.ImpDefSysInstr128()
// =====
// Execute an implementation-defined system instruction with write (128-bit source operand).

AArch64.ImpDefSysInstr128(integer el, bits(3) op1, bits(4) CRn,
                           bits(4) CRm, bits(3) op2,
                           integer t, integer t2);
```



### Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysInstrWithResult

```
// AArch64.ImpDefSysInstrWithResult()
// =====
// Execute an implementation-defined system instruction with read (result operand).

AArch64.ImpDefSysInstrWithResult(integer el, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2);
```

### Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysRegRead

```
// AArch64.ImpDefSysRegRead()
// =====
// Read from an implementation-defined System register and write the contents of the register
// to X[t].

AArch64.ImpDefSysRegRead(bits(2) op0, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2,
                        integer t);
```

### Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysRegRead128

```
// AArch64.ImpDefSysRegRead128()
// =====
// Read from an 128-bit implementation-defined System register
// and write the contents of the register to X[t], X[t+1].

AArch64.ImpDefSysRegRead128(bits(2) op0, bits(3) op1, bits(4) CRn,
                           bits(4) CRm, bits(3) op2,
                           integer t, integer t2);
```

### Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysRegWrite

```
// AArch64.ImpDefSysRegWrite()
// =====
// Write to an implementation-defined System register.

AArch64.ImpDefSysRegWrite(bits(2) op0, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2,
                        integer t);
```

### Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysRegWrite128

```
// AArch64.ImpDefSysRegWrite128()
// =====
// Write the contents of X[t], X[t+1] to an 128-bit implementation-defined System register.

AArch64.ImpDefSysRegWrite128(bits(2) op0, bits(3) op1, bits(4) CRn,
                            bits(4) CRm, bits(3) op2,
                            integer t, integer t2);
```

### Library pseudocode for aarch64/functions/system/AArch64.NextRandomTagBit

```
// AArch64.NextRandomTagBit()
// =====
// Generate a random bit suitable for generating a random Allocation Tag.

bit AArch64.NextRandomTagBit()
    assert GCR_EL1.RRND == '0';
    bits(16) lfsr = RGSRR_EL1.SEED<15:0>;
    bit top = lfsr<5> EOR lfsr<3> EOR lfsr<2> EOR lfsr<0>;
    RGSRR_EL1.SEED<15:0> = top:lfsr<15:1>;
    return top;
```

### Library pseudocode for aarch64/functions/system/AArch64.RandomTag

```
// AArch64.RandomTag()
// =====
// Generate a random Allocation Tag.

bits(4) AArch64.RandomTag()
    bits(4) tag;
    for i = 0 to 3
        tag<i> = AArch64.NextRandomTagBit\(\);
    return tag;
```

### Library pseudocode for aarch64/functions/system/AArch64.SysInstr

```
// AArch64.SysInstr()
// =====
// Execute a system instruction with write (source operand).

AArch64.SysInstr(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);
```

### Library pseudocode for aarch64/functions/system/AArch64.SysInstrWithResult

```
// AArch64.SysInstrWithResult()
// =====
// Execute a system instruction with read (result operand).
// Writes the result of the instruction to X[t].

AArch64.SysInstrWithResult(integer op0, integer op1, integer crn, integer crm, integer op2,
                           integer t);
```

### Library pseudocode for aarch64/functions/system/AArch64.SysRegRead

```
// AArch64.SysRegRead()
// =====
// Read from a System register and write the contents of the register to X[t].

AArch64.SysRegRead(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);
```

### Library pseudocode for aarch64/functions/system/AArch64.SysRegWrite

```
// AArch64.SysRegWrite()
// =====
// Write to a System register.

AArch64.SysRegWrite(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);
```

### Library pseudocode for aarch64/functions/system/BTypeCompatible

```
boolean BTypeCompatible;
```

## Library pseudocode for aarch64/functions/system/BTypeCompatible\_BTI

```
// BTypeCompatible_BTI
// =====
// This function determines whether a given hint encoding is compatible with the current value of
// PSTATE.BTYPE. A value of TRUE here indicates a valid Branch Target Identification instruction.

boolean BTypeCompatible_BTI(bits(2) hintcode)
    case hintcode of
        when '00'
            return FALSE;
        when '01'
            return PSTATE.BTYPE != '11';
        when '10'
            return PSTATE.BTYPE != '10';
        when '11'
            return TRUE;
```

## Library pseudocode for aarch64/functions/system/BTypeCompatible\_PACIXSP

```
// BTypeCompatible_PACIXSP()
// =====
// Returns TRUE if PACIASP, PACIBSP instruction is implicit compatible with PSTATE.BTYPE,
// FALSE otherwise.

boolean BTypeCompatible_PACIXSP()
    if PSTATE.BTYPE IN {'01', '10'} then
        return TRUE;
    elsif PSTATE.BTYPE == '11' then
        index = if PSTATE.EL == ELO then 35 else 36;
        return SCTLR\_ELx[]<index> == '0';
    else
        return FALSE;
```

## Library pseudocode for aarch64/functions/system/BTypeNext

```
bits(2) BTypeNext;
```

## Library pseudocode for aarch64/functions/system/ChooseRandomNonExcludedTag

```
// ChooseRandomNonExcludedTag()
// =====
// The ChooseRandomNonExcludedTag function is used when GCR_EL1.RRND == '1' to generate random
// Allocation Tags.
//
// The resulting Allocation Tag is selected from the set [0,15], excluding any Allocation Tag where
// exclude[tag_value] == 1. If 'exclude' is all Ones, the returned Allocation Tag is '0000'.
//
// This function is permitted to generate a non-deterministic selection from the set of non-excluded
// Allocation Tags. A reasonable implementation is described by the Pseudocode used when
// GCR_EL1.RRND is 0, but with a non-deterministic implementation of NextRandomTagBit().
// Implementations may choose to behave the same as GCR_EL1.RRND=0.
//
// This function can read RGSR_EL1 and/or write RGSR_EL1 to an IMPLEMENTATION DEFINED value.
// If it is not capable of writing RGSR_EL1.SEED[15:0] to zero from a previous nonzero
// RGSR_EL1.SEED value, it is IMPLEMENTATION DEFINED whether the randomness is significantly
// impacted if RGSR_EL1.SEED[15:0] is set to zero.

bits(4) ChooseRandomNonExcludedTag(bits(16) exclude_in);
```

## Library pseudocode for aarch64/functions/system/InGuardedPage

```
boolean InGuardedPage;
```

### Library pseudocode for aarch64/functions/system/IsHCRXEL2Enabled

```
// IsHCRXEL2Enabled()
// =====
// Returns TRUE if access to HCRX_EL2 register is enabled, and FALSE otherwise.
// Indirect read of HCRX_EL2 returns 0 when access is not enabled.

boolean IsHCRXEL2Enabled()
    if !HaveFeatHCRX() then return FALSE;
    if HaveEL(EL3) && SCR_EL3.HXEn == '0' then
        return FALSE;

    return EL2Enabled();
```

### Library pseudocode for aarch64/functions/system/IsSCTLR2EL1Enabled

```
// IsSCTLR2EL1Enabled()
// =====
// Returns TRUE if access to SCTLR2_EL1 register is enabled, and FALSE otherwise.
// Indirect read of SCTLR2_EL1 returns 0 when access is not enabled.

boolean IsSCTLR2EL1Enabled()
    if !HaveFeatSCTLR2() then return FALSE;
    if HaveEL(EL3) && SCR_EL3.SCTLR2En == '0' then
        return FALSE;
    elseif (EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2.SCTLR2En == '0')) then
        return FALSE;
    else
        return TRUE;
```

### Library pseudocode for aarch64/functions/system/IsSCTLR2EL2Enabled

```
// IsSCTLR2EL2Enabled()
// =====
// Returns TRUE if access to SCTLR2_EL2 register is enabled, and FALSE otherwise.
// Indirect read of SCTLR2_EL2 returns 0 when access is not enabled.

boolean IsSCTLR2EL2Enabled()
    if !HaveFeatSCTLR2() then return FALSE;
    if HaveEL(EL3) && SCR_EL3.SCTLR2En == '0' then
        return FALSE;

    return EL2Enabled();
```

### Library pseudocode for aarch64/functions/system/IsTCR2EL1Enabled

```
// IsTCR2EL1Enabled()
// =====
// Returns TRUE if access to TCR2_EL1 register is enabled, and FALSE otherwise.
// Indirect read of TCR2_EL1 returns 0 when access is not enabled.

boolean IsTCR2EL1Enabled()
    if !HaveFeatTCR2() then return FALSE;
    if HaveEL(EL3) && SCR_EL3.TCR2En == '0' then
        return FALSE;
    elseif (EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2.TCR2En == '0')) then
        return FALSE;
    else
        return TRUE;
```

### Library pseudocode for aarch64/functions/system/IsTCR2EL2Enabled

```
// IsTCR2EL2Enabled()
// =====
// Returns TRUE if access to TCR2_EL2 register is enabled, and FALSE otherwise.
// Indirect read of TCR2_EL2 returns 0 when access is not enabled.

boolean IsTCR2EL2Enabled()
    if !HaveFeatTCR2() then return FALSE;
    if HaveEL(EL3) && SCR_EL3.TCR2En == '0' then
        return FALSE;

    return EL2Enabled();
```

### Library pseudocode for aarch64/functions/system/SetBTypeCompatible

```
// SetBTypeCompatible()
// =====
// Sets the value of BTypeCompatible global variable used by BTI

SetBTypeCompatible(boolean x)
    BTypeCompatible = x;
```

### Library pseudocode for aarch64/functions/system/SetBTypeNext

```
// SetBTypeNext()
// =====
// Set the value of BTypeNext global variable used by BTI

SetBTypeNext(bits(2) x)
    BTypeNext = x;
```

### Library pseudocode for aarch64/functions/system/SetInGuardedPage

```
// SetInGuardedPage()
// =====
// Global state updated to denote if memory access is from a guarded page.

SetInGuardedPage(boolean guardedpage)
    InGuardedPage = guardedpage;
```

### Library pseudocode for aarch64/functions/system128/AArch64.SysInstr128

```
// AArch64.SysInstr128()
// =====
// Execute a system instruction with write (2 64-bit source operands).

AArch64.SysInstr128(integer op0, integer op1, integer crn, integer crm,
                    integer op2, integer t, integer t2);
```

### Library pseudocode for aarch64/functions/system128/AArch64.SysRegRead128

```
// AArch64.SysRegRead128()
// =====
// Read from a 128-bit System register and write the contents of the register to X[t] and X[t2].

AArch64.SysRegRead128(integer op0, integer op1, integer crn, integer crm,
                    integer op2, integer t, integer t2);
```

## Library pseudocode for aarch64/functions/system128/AArch64.SysRegWrite128

```
// AArch64.SysRegWrite128()
// =====
// Read the contents of X[t] and X[t2] and write the contents to a 128-bit System register.

AArch64.SysRegWrite128(integer op0, integer op1, integer crn, integer crm,
                      integer op2, integer t, integer t2);
```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBIP\_IPAS2

```
// AArch64.TLBIP_IPAS2()
// =====
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Xt.

AArch64.TLBIP_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                   Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(128) Xt)
assert PSTATE.EL IN {EL3, EL2};

TLBIRecord r;
r.op = TLBIOp\_IPAS2;
r.from_aarch64 = TRUE;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = level;
r.attr = attr;
r.ttl = Xt<47:44>;
r.address = ZeroExtend(Xt<107:64> : Zeros(12), 64);
r.d64 = r.ttl IN {'00xx'};
r.d128 = TRUE;

case security of
  when SS\_NonSecure
    r.ipaspace = PAS\_NonSecure;
  when SS\_Secure
    r.ipaspace = if Xt<63> == '1' then PAS\_NonSecure else PAS\_Secure;
  when SS\_Realm
    r.ipaspace = PAS\_Realm;
  otherwise
    // Root security state does not have stage 2 translation
    Unreachable();

TLBI(r);
if shareability != Shareability\_NSH then Broadcast(shareability, r);
return;
```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBIP\_RIPAS2

```
// AArch64.TLBIP_RIPAS2()
// =====
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// shareability domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.

AArch64.TLBIP_RIPAS2(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_RIPAS2;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.ttl<1:0> = Xt<38:37>;
    r.d64 = r.ttl<1:0> == '00';
    r.d128 = TRUE;

    bits(2) tg = Xt<47:46>;
    integer scale = UInt(Xt<45:44>);
    integer num = UInt(Xt<43:39>);
    integer baseaddr = SInt(Xt<36:0>);

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

    if !valid then return;

    case security of
        when SS\_NonSecure
            r.ipaspace = PAS\_NonSecure;
        when SS\_Secure
            r.ipaspace = if Xt<63> == '1' then PAS\_NonSecure else PAS\_Secure;
        when SS\_Realm
            r.ipaspace = PAS\_Realm;
        otherwise
            // Root security state does not have stage 2 translation
            Unreachable();

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBIP\_RVA

```
// AArch64.TLBIP_RVA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID and ASID (where regime
// supports VMID, ASID) in the indicated regime with the indicated security state.
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBIP_RVA(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_RVA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid = Xt<63:48>;
    r.ttl<1:0> = Xt<38:37>;
    r.d64 = r.ttl<1:0> == '00';
    r.d128 = TRUE;

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```



## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBIP\_RVAA

```
// AArch64.TLBIP_RVAA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID (where regimesupports VMID)
// and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBIP_RVAA(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_RVAA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.ttl<1:0> = Xt<38:37>;
    r.d64 = r.ttl<1:0> == '00';
    r.d128 = TRUE;

    bits(2) tg = Xt<47:46>;
    integer scale = UInt(Xt<45:44>);
    integer num = UInt(Xt<43:39>);
    integer baseaddr = SInt(Xt<36:0>);

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBIP\_VA

```
// AArch64.TLBIP_VA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBIP_VA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_VA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid = Xt<63:48>;
    r.ttl = Xt<47:44>;
    r.address = ZeroExtend(Xt<107:64> : Zeros(12), 64);
    r.d64 = r.ttl IN {'00xx'};
    r.d128 = TRUE;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBIP\_VAA

```
// AArch64.TLBIP_VAA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBIP_VAA(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_VAA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.ttl = Xt<47:44>;
    r.address = ZeroExtend(Xt<107:64> : Zeros(12), 64);
    r.d64 = r.ttl IN {'00xx'};
    r.d128 = TRUE;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_ALL

```
// AArch64.TLBI_ALL()
// =====
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

AArch64.TLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op = TLBIOp\_ALL;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.level = TLBILevel\_Any;
    r.attr = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_ASID

```
// AArch64.TLBI_ASID()
// =====
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Xt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_ASID;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = TLBILevel\_Any;
    r.attr = attr;
    r.asid = Xt<63:48>;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_IPAS2

```
// AArch64.TLBI_IPAS2()
// =====
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Xt.

AArch64.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                   Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op          = TLBIOp\_IPAS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid        = vmid;
    r.level       = level;
    r.attr        = attr;
    r.ttl         = Xt<47:44>;
    r.address     = ZeroExtend(Xt<39:0> : Zeros(12), 64);
    r.d64         = TRUE;
    r.d128        = r.ttl IN {'00xx'};

    case security of
        when SS\_NonSecure
            r.ipaspace = PAS\_NonSecure;
        when SS\_Secure
            r.ipaspace = if Xt<63> == '1' then PAS\_NonSecure else PAS\_Secure;
        when SS\_Realm
            r.ipaspace = PAS\_Realm;
        otherwise
            // Root security state does not have stage 2 translation
            Unreachable();

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_PAALL

```
// AArch64.TLBI_PAALL()
// =====
// TLB Invalidate ALL GPT Information.
// Invalidates cached copies of GPT entries from TLBs in the indicated
// Shareability domain.
// The invalidation applies to all TLB entries containing GPT information.

AArch64.TLBI_PAALL(Shareability shareability)
    assert HaveRME() && PSTATE.EL == EL3;

    TLBIRecord r;

    // r.security and r.regime do not apply for TLBI by PA operations
    r.op      = TLBIOp\_PAALL;
    r.level   = TLBILevel\_Any;
    r.attr    = TLBI\_AllAttr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);

    return;
```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_RIPAS2

```
// AArch64.TLBI_RIPAS2()
// =====
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// shareability domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.

AArch64.TLBI_RIPAS2(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_RIPAS2;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.ttl<1:0> = Xt<38:37>;
    r.d64 = TRUE;
    r.d128 = r.ttl<1:0> == '00';

    bits(2) tg = Xt<47:46>;
    integer scale = UInt(Xt<45:44>);
    integer num = UInt(Xt<43:39>);
    integer baseaddr = SInt(Xt<36:0>);

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    case security of
        when SS\_NonSecure
            r.ipaspace = PAS\_NonSecure;
        when SS\_Secure
            r.ipaspace = if Xt<63> == '1' then PAS\_NonSecure else PAS\_Secure;
        when SS\_Realm
            r.ipaspace = PAS\_Realm;
        otherwise
            // Root security state does not have stage 2 translation
            Unreachable();

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_RPA

```
// AArch64.TLBI_RPA()
// =====
// TLB Range Invalidate GPT Information by PA.
// Invalidates cached copies of GPT entries from TLBs in the indicated
// Shareability domain.
// The invalidation applies to TLB entries containing GPT information relating
// to the indicated physical address range.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries containing GPT information
//                     from all levels of the GPT walk
//     TLBILevel_Last : this applies to TLB entries containing GPT information
//                     from the last level of the GPT walk

AArch64.TLBI_RPA(TLBILevel level, bits(64) Xt, Shareability shareability)
    assert HaveRME() && PSTATE.EL == EL3;

    TLBIRecord r;
    integer range_bits;
    integer p;

    // r.security and r.regime do not apply for TLBI by PA operations
    r.op    = TLBIOp\_RPA;
    r.level = level;
    r.attr  = TLBI\_AllAttr;

    // SIZE field
    case Xt<47:44> of
        when '0000' range_bits = 12; // 4KB
        when '0001' range_bits = 14; // 16KB
        when '0010' range_bits = 16; // 64KB
        when '0011' range_bits = 21; // 2MB
        when '0100' range_bits = 25; // 32MB
        when '0101' range_bits = 29; // 512MB
        when '0110' range_bits = 30; // 1GB
        when '0111' range_bits = 34; // 16GB
        when '1000' range_bits = 36; // 64GB
        when '1001' range_bits = 39; // 512GB
        otherwise range_bits = 0; // Reserved encoding

    // If SIZE selects a range smaller than PGS, then PGS is used instead
    case DecodePGS(GPCCR_EL3.PGS) of
        when PGS\_4KB p = 12;
        when PGS\_16KB p = 14;
        when PGS\_64KB p = 16;

    if range_bits < p then
        range_bits = p;

    bits(52) BaseADDR = Zeros(52);
    case GPCCR_EL3.PGS of
        when '00' BaseADDR<51:12> = Xt<39:0>; // 4KB
        when '10' BaseADDR<51:14> = Xt<39:2>; // 16KB
        when '01' BaseADDR<51:16> = Xt<39:4>; // 64KB

    // The calculation here automatically aligns BaseADDR to the size of
    // the region specified in SIZE. However, the architecture does not
    // require this alignment and if BaseADDR is not aligned to the region
    // specified by SIZE then no entries are required to be invalidated.
    bits(52) start_addr = BaseADDR AND NOT ZeroExtend(Ones(range_bits), 52);
    bits(52) end_addr   = start_addr + ZeroExtend(Ones(range_bits), 52);

    // PASpace is not considered in TLBI by PA operations
    r.address    = ZeroExtend(start_addr, 64);
    r.end_address = ZeroExtend(end_addr, 64);

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_RVA

```
// AArch64.TLBI_RVA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID and ASID (where regime
// supports VMID, ASID) in the indicated regime with the indicated security state.
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_RVA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_RVA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid = Xt<63:48>;
    r.ttl<1:0> = Xt<38:37>;
    r.d64 = TRUE;
    r.d128 = r.ttl<1:0> == '00';

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_RVAA

```
// AArch64.TLBI_RVAA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID (where regimesupports VMID)
// and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_RVAA(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_RVAA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.ttl<1:0> = Xt<38:37>;
    r.d64 = TRUE;
    r.d128 = r.ttl<1:0> == '00';

    bits(2) tg = Xt<47:46>;
    integer scale = UInt(Xt<45:44>);
    integer num = UInt(Xt<43:39>);
    integer baseaddr = SInt(Xt<36:0>);

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```



## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_VA

```
// AArch64.TLBI_VA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
               Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_VA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid = Xt<63:48>;
    r.ttl = Xt<47:44>;
    r.address = ZeroExtend(Xt<43:0> : Zeros(12), 64);
    r.d64 = TRUE;
    r.d128 = r.ttl IN {'00xx'};

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_VAA

```
// AArch64.TLBI_VAA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
                Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_VAA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.ttl = Xt<47:44>;
    r.address = ZeroExtend(Xt<43:0> : Zeros(12), 64);
    r.d64 = TRUE;
    r.d128 = r.ttl IN {'00xx'};

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_VMALL

```
// AArch64.TLBI_VMALL()
// =====
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.

AArch64.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_VMALL;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.level = TLBILevel\_Any;
    r.vmid = vmid;
    r.attr = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_VMALLS12

```
// AArch64.TLBI_VMALLS12()
// =====
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.

AArch64.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op = TLBIOp\_VMALLS12;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.level = TLBILevel\_Any;
    r.vmid = vmid;
    r.attr = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

## Library pseudocode for aarch64/functions/tlbi/ASID\_NONE

```
constant bits(16) ASID_NONE = Zeros(16);
```

## Library pseudocode for aarch64/functions/tlbi/Broadcast

```
// Broadcast()
// =====
// IMPLEMENTATION DEFINED function to broadcast TLBI operation within the indicated shareability
// domain.

Broadcast(Shareability shareability, TLBIRecord r)
    IMPLEMENTATION_DEFINED;
```

## Library pseudocode for aarch64/functions/tlbi/DecodeTLBITG

```
// DecodeTLBITG()
// =====
// Decode translation granule size in TLBI range instructions

TGx DecodeTLBITG(bits(2) tg)
  case tg of
    when '01'   return TGx\_4KB;
    when '10'   return TGx\_16KB;
    when '11'   return TGx\_64KB;
```

## Library pseudocode for aarch64/functions/tlbi/GPTTLBIMatch

```
// GPTTLBIMatch()
// =====
// Determine whether the GPT TLB entry lies within the scope of invalidation

boolean GPTTLBIMatch(TLBIRecord tlbi, GPTEntry gpt_entry)
  assert tlbi.op IN {TLBIOp\_RPA, TLBIOp\_PAALL};

  boolean match;
  bits(64) entry_size_mask      = ZeroExtend(Ones(gpt_entry.size), 64);
  bits(64) entry_end_address    = ZeroExtend(gpt_entry.pa<55:0> OR entry_size_mask<55:0>, 64);
  bits(64) entry_start_address = ZeroExtend(gpt_entry.pa<55:0> AND NOT entry_size_mask<55:0>, 64);

  case tlbi.op of
    when TLBIOp\_RPA
      match = (UInt(tlbi.address<55:0>) <= UInt(entry_end_address<55:0>) &&
               UInt(tlbi.end_address<55:0>) > UInt(entry_start_address<55:0>) &&
               (tlbi.level == TLBILevel\_Any || gpt_entry.level == 1));
    when TLBIOp\_PAALL
      match = TRUE;

  return match;
```

## Library pseudocode for aarch64/functions/tlbi/HasLargeAddress

```
// HasLargeAddress()
// =====
// Returns TRUE if the regime is configured for 52 bit addresses, FALSE otherwise.

boolean HasLargeAddress(Regime regime)
  if !Have52BitIPAAndPASpaceExt() then
    return FALSE;
  case regime of
    when Regime\_EL3
      return TCR_EL3<32> == '1';
    when Regime\_EL2
      return TCR_EL2<32> == '1';
    when Regime\_EL20
      return TCR_EL2<59> == '1';
    when Regime\_EL10
      return TCR_EL1<59> == '1';
    otherwise
      Unreachable();
```

### Library pseudocode for aarch64/functions/tlbi/ResTLBIRTTL

```
// ResTLBIRTTL()
// =====
// Determine whether the TTL field in TLBI instructions that do apply
// to a range of addresses contains a reserved value

boolean ResTLBIRTTL(bits(2) tg, bits(2) ttl)
    case ttl of
        when '00' return TRUE;
        when '01' return DecodeTLBITG(tg) == TGx\_16KB && !Have52BitIPAAAndPASpaceExt();
        otherwise return FALSE;
```

### Library pseudocode for aarch64/functions/tlbi/ResTLBITTL

```
// ResTLBITTL()
// =====
// Determine whether the TTL field in TLBI instructions that do not apply
// to a range of addresses contains a reserved value

boolean ResTLBITTL(bits(4) ttl)
    case ttl of
        when '00xx' return TRUE;
        when '0100' return !Have52BitIPAAAndPASpaceExt();
        when '1000' return TRUE;
        when '1001' return !Have52BitIPAAAndPASpaceExt();
        when '1100' return TRUE;
        otherwise return FALSE;
```

### Library pseudocode for aarch64/functions/tlbi/TLBI

```
// TLBI()
// =====
// Invalidates TLB entries for which TLBIMatch() returns TRUE.

TLBI(TLBIRecord r)
    IMPLEMENTATION_DEFINED;
```

### Library pseudocode for aarch64/functions/tlbi/TLBILevel

```
// TLBILevel
// =====

enumeration TLBILevel {
    TLBILevel_Any,          // this applies to TLB entries at all levels
    TLBILevel_Last         // this applies to TLB entries at last level only
};
```



```

// TLBIMatch()
// =====
// Determine whether the TLB entry lies within the scope of invalidation

boolean TLBIMatch(TLBIRecord tlbi, TLBRecord tlb_entry)
    boolean match;
    bits(64) entry_block_mask    = ZeroExtend(Ones(tlb_entry.blocksize), 64);
    bits(64) entry_end_address    = tlb_entry.context.ia OR entry_block_mask;
    bits(64) entry_start_address = tlb_entry.context.ia AND NOT entry_block_mask;
    case tlbi.op of
        when TLBIOp\_DALL, TLBIOp\_IALL
            match = (tlbi.security == tlb_entry.context.ss &&
                    tlb_entry.regime == tlb_entry.context.regime);
        when TLBIOp\_DASID, TLBIOp\_IASID
            match = (tlb_entry.context.includes_s1 &&
                    tlb_entry.security == tlb_entry.context.ss &&
                    tlb_entry.regime == tlb_entry.context.regime &&
                    (!UseVMID(tlb_entry.context) || tlb_entry.vmid == tlb_entry.context.vmid) &&
                    (UseASID(tlb_entry.context) && tlb_entry.context.nG == '1' &&
                     tlb_entry.asid == tlb_entry.context.asid));
        when TLBIOp\_DVA, TLBIOp\_IVA
            boolean regime_match;
            boolean context_match;
            boolean address_match;
            boolean level_match;
            regime_match = (tlb_entry.context.includes_s1 &&
                           tlb_entry.security == tlb_entry.context.ss &&
                           tlb_entry.regime == tlb_entry.context.regime);
            context_match = ((!UseVMID(tlb_entry.context) || tlb_entry.vmid == tlb_entry.context.vmid) &&
                           (!UseASID(tlb_entry.context) || tlb_entry.asid == tlb_entry.context.asid ||
                            tlb_entry.context.nG == '0'));
            integer addr_lsb = tlb_entry.blocksize;
            address_match = tlb_entry.address<55:addr_lsb> == tlb_entry.context.ia<55:addr_lsb>;
            level_match = (tlb_entry.level == TLBILevel\_Any || !tlb_entry.walkstate.istable);
            match = regime_match && context_match && address_match && level_match;
        when TLBIOp\_ALL
            relax_regime = (tlbi.from_aarch64 &&
                           tlb_entry.regime IN {Regime\_EL20, Regime\_EL2} &&
                           tlb_entry.context.regime IN {Regime\_EL20, Regime\_EL2});
            match = (tlbi.security == tlb_entry.context.ss &&
                    (tlbi.regime == tlb_entry.context.regime || relax_regime));
        when TLBIOp\_ASID
            match = (tlb_entry.context.includes_s1 &&
                    tlb_entry.security == tlb_entry.context.ss &&
                    tlb_entry.regime == tlb_entry.context.regime &&
                    (!UseVMID(tlb_entry.context) || tlb_entry.vmid == tlb_entry.context.vmid) &&
                    (UseASID(tlb_entry.context) && tlb_entry.context.nG == '1' &&
                     tlb_entry.asid == tlb_entry.context.asid));
        when TLBIOp\_IPAS2, TLBIOp\_IPAS2
            integer addr_lsb = tlb_entry.blocksize;
            match = (!tlb_entry.context.includes_s1 && tlb_entry.context.includes_s2 &&
                    tlb_entry.security == tlb_entry.context.ss &&
                    tlb_entry.regime == tlb_entry.context.regime &&
                    (!UseVMID(tlb_entry.context) || tlb_entry.vmid == tlb_entry.context.vmid) &&
                    tlb_entry.ipaspace == tlb_entry.context.ipaspace &&
                    tlb_entry.address<55:addr_lsb> == tlb_entry.context.ia<55:addr_lsb> &&
                    (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
                        DecodeTLBITG(tlbi.ttl<3:2>) == tlb_entry.context.tg &&
                        UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level)
                    ) &&
                    ((tlbi.d128 && tlb_entry.context.isd128) ||
                     (tlbi.d64 && !tlb_entry.context.isd128) ||
                     (tlbi.d64 && tlb_entry.d128)) &&
                    (tlb_entry.level == TLBILevel\_Any || !tlb_entry.walkstate.istable));
        when TLBIOp\_VAA, TLBIOp\_VAA
            integer addr_lsb = tlb_entry.blocksize;
            match = (tlb_entry.context.includes_s1 &&
                    tlb_entry.security == tlb_entry.context.ss &&
                    tlb_entry.regime == tlb_entry.context.regime &&
                    (!UseVMID(tlb_entry.context) || tlb_entry.vmid == tlb_entry.context.vmid) &&

```

```

        tlb_entry.context.ia<55:addr_lsb> == tlb_entry.context.ia<55:addr_lsb> &&
        (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
            DecodeTLBITG(tlbi.ttl<3:2>) == tlb_entry.context.tg &&
            UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level)
        ) &&
        ((tlbi.d128 && tlb_entry.context.isd128) ||
         (tlbi.d64 && !tlb_entry.context.isd128) ||
         (tlbi.d64 && tlbi.d128)) &&
        (tlbi.level == TLBIlevel Any || !tlb_entry.walkstate.istable));
when TLBIOp VA, TLBIOp VA
    integer addr_lsb = tlb_entry.blocksize;
    match = (tlb_entry.context.includes_s1 &&
        tlb_entry.security == tlb_entry.context.ss &&
        tlb_entry.regime == tlb_entry.context.regime &&
        (!UseVMID(tlb_entry.context) || tlb_entry.vmid == tlb_entry.context.vmid) &&
        (!UseASID(tlb_entry.context) || tlb_entry.asid == tlb_entry.context.asid ||
         tlb_entry.context.nG == '0') &&
        tlb_entry.address<55:addr_lsb> == tlb_entry.context.ia<55:addr_lsb> &&
        (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
            DecodeTLBITG(tlbi.ttl<3:2>) == tlb_entry.context.tg &&
            UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level)
        ) &&
        ((tlbi.d128 && tlb_entry.context.isd128) ||
         (tlbi.d64 && !tlb_entry.context.isd128) ||
         (tlbi.d64 && tlbi.d128)) &&
        (tlbi.level == TLBIlevel Any || !tlb_entry.walkstate.istable));
when TLBIOp VMALL
    match = (tlb_entry.context.includes_s1 &&
        tlb_entry.security == tlb_entry.context.ss &&
        tlb_entry.regime == tlb_entry.context.regime &&
        (!UseVMID(tlb_entry.context) || tlb_entry.vmid == tlb_entry.context.vmid));
when TLBIOp VMALLS12
    match = (tlbi.security == tlb_entry.context.ss &&
        tlb_entry.regime == tlb_entry.context.regime &&
        (!UseVMID(tlb_entry.context) || tlb_entry.vmid == tlb_entry.context.vmid));
when TLBIOp RIPAS2, TLBIOp RIPAS2
    match = (!tlb_entry.context.includes_s1 && tlb_entry.context.includes_s2 &&
        tlb_entry.security == tlb_entry.context.ss &&
        tlb_entry.regime == tlb_entry.context.regime &&
        (!UseVMID(tlb_entry.context) || tlb_entry.vmid == tlb_entry.context.vmid) &&
        tlb_entry.ipaspace == tlb_entry.context.ipaspace &&
        (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == tlb_entry.context.tg) &&
        (!tlbi.from_aarch64 || ResTLBITRTTL(tlbi.tg, tlb_entry.ttl<1:0>) ||
         UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level) &&
        ((tlbi.d128 && tlb_entry.context.isd128) ||
         (tlbi.d64 && !tlb_entry.context.isd128) ||
         (tlbi.d64 && tlbi.d128)) &&
        UInt(tlbi.address<55:0>) <= UInt(entry_end_address<55:0>) &&
        UInt(tlbi.end_address<55:0>) > UInt(entry_start_address<55:0>));
when TLBIOp RVAA, TLBIOp RVAA
    match = (tlb_entry.context.includes_s1 &&
        tlb_entry.security == tlb_entry.context.ss &&
        tlb_entry.regime == tlb_entry.context.regime &&
        (!UseVMID(tlb_entry.context) || tlb_entry.vmid == tlb_entry.context.vmid) &&
        (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == tlb_entry.context.tg) &&
        (!tlbi.from_aarch64 || ResTLBITRTTL(tlbi.tg, tlb_entry.ttl<1:0>) ||
         UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level) &&
        ((tlbi.d128 && tlb_entry.context.isd128) ||
         (tlbi.d64 && !tlb_entry.context.isd128) ||
         (tlbi.d64 && tlbi.d128)) &&
        UInt(tlbi.address<55:0>) <= UInt(entry_end_address<55:0>) &&
        UInt(tlbi.end_address<55:0>) > UInt(entry_start_address<55:0>));
when TLBIOp RVA, TLBIOp RVA
    match = (tlb_entry.context.includes_s1 &&
        tlb_entry.security == tlb_entry.context.ss &&
        tlb_entry.regime == tlb_entry.context.regime &&
        (!UseVMID(tlb_entry.context) || tlb_entry.vmid == tlb_entry.context.vmid) &&
        (!UseASID(tlb_entry.context) || tlb_entry.asid == tlb_entry.context.asid ||
         tlb_entry.context.nG == '0') &&
        (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == tlb_entry.context.tg) &&

```

```

        (!tlbi.from_aarch64 || ResTLBIRTTL(tlbi.tg, tlbi.ttl<1:0>) ||
         UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level) &&
        ((tlbi.d128 && tlb_entry.context.isd128) ||
         (tlbi.d64 && !tlb_entry.context.isd128) ||
         (tlbi.d64 && tlbi.d128)) &&
        UInt(tlbi.address<55:0>) <= UInt(entry_end_address<55:0>) &&
        UInt(tlbi.end_address<55:0>) > UInt(entry_start_address<55:0>));
when TLBIOp\_RPA
    entry_end_address<55:0> = (tlb_entry.walkstate.baseaddress.address<55:0> OR
                             entry_block_mask<55:0>);
    entry_start_address<55:0> = (tlb_entry.walkstate.baseaddress.address<55:0> AND
                                NOT entry_block_mask<55:0>);
    match = (tlb_entry.context.includes_gpt &&
             UInt(tlbi.address<55:0>) <= UInt(entry_end_address<55:0>) &&
             UInt(tlbi.end_address<55:0>) > UInt(entry_start_address<55:0>));
when TLBIOp\_PAALL
    match = tlb_entry.context.includes_gpt;

if tlbi.attr == TLBI\_ExcludeXS && tlb_entry.context.xs == '1' then
    match = FALSE;

return match;

```

### Library pseudocode for aarch64/functions/tlbi/TLBIMemAttr

```

// TLBIMemAttr
// =====
// Defines the attributes of the memory operations that must be completed in
// order to deem the TLBI operation as completed.

enumeration TLBIMemAttr {
    TLBI_AllAttr,           // All TLB entries within the scope of the invalidation
    TLBI_ExcludeXS         // Only TLB entries with XS=0 within the scope of the invalidation
};

```

### Library pseudocode for aarch64/functions/tlbi/TLBIOp

```

// TLBIOp
// =====

enumeration TLBIOp {
    TLBIOp_DALL,           // AArch32 Data TLBI operations - deprecated
    TLBIOp_DASID,
    TLBIOp_DVA,
    TLBIOp_IALL,           // AArch32 Instruction TLBI operations - deprecated
    TLBIOp_IASID,
    TLBIOp_IVA,
    TLBIOp_ALL,
    TLBIOp_ASID,
    TLBIOp_IPAS2,
    TLBIOp_IPAS2,
    TLBIOp_VAA,
    TLBIOp_VA,
    TLBIOp_VAA,
    TLBIOp_VA,
    TLBIOp_VMALL,
    TLBIOp_VMALLS12,
    TLBIOp_RIPAS2,
    TLBIOp_RIPAS2,
    TLBIOp_RVAA,
    TLBIOp_RVA,
    TLBIOp_RVAA,
    TLBIOp_RVA,
    TLBIOp_RPA,
    TLBIOp_PAALL,
};

```



## Library pseudocode for aarch64/functions/tlbi/TLBIPRange

```
// TLBIPRange()
// =====
// Extract the input address range information from encoded Xt.

(boolean, bits(2), bits(64), bits(64)) TLBIPRange(Regime regime, bits(128) Xt)
    boolean valid      = TRUE;
    bits(64) start_address = Zeros(64);
    bits(64) end_address  = Zeros(64);

    bits(2) tg          = Xt<47:46>;
    integer scale        = UInt(Xt<45:44>);
    integer num          = UInt(Xt<43:39>);
    integer tg_bits;

    if tg == '00' then
        return (FALSE, tg, start_address, end_address);

    case tg of
        when '01' // 4KB
            tg_bits = 12;
            start_address<55:12> = Xt<107:64>;
            start_address<63:56> = Replicate(Xt<107>, 8);
        when '10' // 16KB
            tg_bits = 14;
            start_address<55:14> = Xt<107:66>;
            start_address<63:56> = Replicate(Xt<107>, 8);
        when '11' // 64KB
            tg_bits = 16;
            start_address<55:16> = Xt<107:68>;
            start_address<63:56> = Replicate(Xt<107>, 8);
        otherwise
            Unreachable();

    integer range = (num+1) << (5*scale + 1 + tg_bits);
    end_address  = start_address + range<63:0>;

    if end_address<55> != start_address<55> then
        // overflow, saturate it
        end_address = Replicate(start_address<55>, 64-55) : Ones(55);

    return (valid, tg, start_address, end_address);
```

## Library pseudocode for aarch64/functions/tlbi/TLBIRange

```
// TLBIRange()
// =====
// Extract the input address range information from encoded Xt.

(boolean, bits(2), bits(64), bits(64)) TLBIRange(Regime regime, bits(64) Xt)
    boolean valid = TRUE;
    bits(64) start_address = Zeros(64);
    bits(64) end_address   = Zeros(64);

    bits(2) tg          = Xt<47:46>;
    integer scale       = UInt(Xt<45:44>);
    integer num         = UInt(Xt<43:39>);
    integer tg_bits;

    if tg == '00' then
        return (FALSE, tg, start_address, end_address);

    case tg of
        when '01' // 4KB
            tg_bits = 12;
            if HasLargeAddress(regime) then
                start_address<52:16> = Xt<36:0>;
                start_address<63:53> = Replicate(Xt<36>, 11);
            else
                start_address<48:12> = Xt<36:0>;
                start_address<63:49> = Replicate(Xt<36>, 15);
        when '10' // 16KB
            tg_bits = 14;
            if HasLargeAddress(regime) then
                start_address<52:16> = Xt<36:0>;
                start_address<63:53> = Replicate(Xt<36>, 11);
            else
                start_address<50:14> = Xt<36:0>;
                start_address<63:51> = Replicate(Xt<36>, 13);
        when '11' // 64KB
            tg_bits = 16;
            start_address<52:16> = Xt<36:0>;
            start_address<63:53> = Replicate(Xt<36>, 11);
        otherwise
            Unreachable();

    integer range = (num+1) << (5*scale + 1 + tg_bits);
    end_address   = start_address + range<63:0>;

    if end_address<52> != start_address<52> then
        // overflow, saturate it
        end_address = Replicate(start_address<52>, 64-52) : Ones(52);

    return (valid, tg, start_address, end_address);
```

## Library pseudocode for aarch64/functions/tlbi/TLBIRecord

```
// TLBIRecord
// =====
// Details related to a TLBI operation.

type TLBIRecord is (
    TLBIOp          op,
    boolean         from_aarch64, // originated as an AArch64 operation
    SecurityState  security,
    Regime          regime,
    bits(16)        vmid,
    bits(16)        asid,
    TLBILevel      level,
    TLBIMemAttr    attr,
    PASpace        ipaspace,      // For operations that take IPA as input address
    bits(64)        address,      // input address, for range operations, start address
    bits(64)        end_address,  // for range operations, end address
    boolean         d64,          // For operations that evict VMSAv8-64 based TLB entries
    boolean         d128,         // For operations that evict VMSAv9-128 based TLB entries
    bits(4)         ttl,          // translation table walk level holding the leaf entry
                                // for the address being invalidated
                                // For Non-Range Invalidations:
                                //   When the ttl is
                                //   '00xx' : this applies to all TLB entries
                                //   Otherwise : TLBIP instructions invalidates D128 TLB
                                //           entries only
                                //           TLBI instructions invalidates D64 TLB
                                //           entries only
                                // For Range Invalidations:
                                //   When the ttl is
                                //   '00' : this applies to all TLB entries
                                //   Otherwise : TLBIP instructions invalidates D128 TLB
                                //           entries only
                                //           TLBI instructions invalidates D64 TLB
                                //           entries only
    bits(2)         tg            // for range operations, translation granule
)
```

## Library pseudocode for aarch64/functions/tlbi/VMID

```
// VMID[]
// =====
// Effective VMID.

bits(16) VMID[]
    if EL2Enabled() then
        if !ELUsingAArch32(EL2) then
            if Have16bitVMID() && VTCR_EL2.VS == '1' then
                return VTTBR_EL2.VMID;
            else
                return ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        else
            return ZeroExtend(VTTBR.VMID, 16);
    elsif HaveEL(EL2) && HaveSecureEL2Ext() then
        return Zeros(16);
    else
        return VMID\_NONE;
```

## Library pseudocode for aarch64/functions/tlbi/VMID\_NONE

```
constant bits(16) VMID_NONE = Zeros(16);
```

## Library pseudocode for aarch64/functions/tme/CheckTransactionalSystemAccess

```
// CheckTransactionalSystemAccess()
// =====
// Returns TRUE if an AArch64 MSR, MRS, or SYS instruction is permitted in
// Transactional state, based on the opcode's encoding, and FALSE otherwise.

boolean CheckTransactionalSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm,
                                      bits(3) op2, bit read)

    case read:op0:op1:crn:crm:op2 of
        when '0 00 011 0100 xxxx 11x' return TRUE;           // MSR (imm): DAIFSet, DAIFClr
        when '0 01 011 0111 0100 001' return TRUE;           // DC ZVA
        when '0 11 011 0100 0010 00x' return TRUE;           // MSR: NZCV, DAIF
        when '0 11 011 0100 0100 00x' return TRUE;           // MSR: FPCR, FPSR
        when '0 11 000 0100 0110 000' return TRUE;           // MSR: ICC_PMR_EL1
        when '0 11 011 1001 1100 100' return TRUE;           // MRS: PMSWINC_EL0
        when '1 11 011 0010 0101 001' return TRUE;           // MRS: GCSPR_EL0, at EL0
        return PSTATE.EL == EL0;
    // MRS: GCSPR_EL1 at EL1 OR at EL2 when E2H is '1'
    when '1 11 000 0010 0101 001'
        return PSTATE.EL == EL1 || (PSTATE.EL == EL2 && HCR_EL2.E2H == '1');
    when '1 11 100 0010 0101 001'
        return PSTATE.EL == EL2 && HCR_EL2.E2H == '0';
    when '1 11 110 0010 0101 001'
        return PSTATE.EL == EL3;
    when '0 01 011 0111 0111 000' return TRUE;               // GCSPUSHM
    when '1 01 011 0111 0111 001' return TRUE;               // GCSPOPM
    when '0 01 011 0111 0111 010' return TRUE;               // GCSSS1
    when '1 01 011 0111 0111 011' return TRUE;               // GCSSS2
    when '0 01 000 0111 0111 110' return TRUE;               // GCSPOPX
    when '1 11 101 0010 0101 001' return FALSE;              // MRS: GCSPR_EL12
    when '1 11 000 0010 0101 010' return FALSE;              // MRS: GCSCRE0_EL1
    when '1 11 000 0010 0101 000' return FALSE;              // MRS: GCSCR_EL1
    when '1 11 101 0010 0101 000' return FALSE;              // MRS: GCSCR_EL12
    when '1 11 100 0010 0101 000' return FALSE;              // MRS: GCSCR_EL2
    when '1 11 110 0010 0101 000' return FALSE;              // MRS: GCSCR_EL3
    when '1 11 xxx 0xxx xxxx xxx' return TRUE;               // MRS: op0=3, CRn=0..7
    when '1 11 xxx 100x xxxx xxx' return TRUE;               // MRS: op0=3, CRn=8..9
    when '1 11 xxx 1010 xxxx xxx' return TRUE;               // MRS: op0=3, CRn=10
    when '1 11 000 1100 1x00 010' return TRUE;               // MRS: op0=3, CRn=12 - ICC_HPPIRx_EL1
    when '1 11 000 1100 1011 011' return TRUE;               // MRS: op0=3, CRn=12 - ICC_RPR_EL1
    when '1 11 xxx 1101 xxxx xxx' return TRUE;               // MRS: op0=3, CRn=13
    when '1 11 xxx 1110 xxxx xxx' return TRUE;               // MRS: op0=3, CRn=14
    when '0 01 011 0111 0011 111' return TRUE;               // CPP RCTX
    when '0 01 011 0111 0011 10x' return TRUE;               // CFP RCTX, DVP RCTX
    when 'x 11 xxx 1x11 xxxx xxx' return TRUE;               // MRS: op1=3, CRn=11,15
    return boolean IMPLEMENTATION_DEFINED;
    otherwise return FALSE;                                   // All other SYS, SYSL, MRS, MSR
```

## Library pseudocode for aarch64/functions/tme/CommitTransactionalWrites

```
// CommitTransactionalWrites()
// =====
// Makes all transactional writes to memory observable by other PEs and reset
// the transactional read and write sets.

CommitTransactionalWrites();
```

## Library pseudocode for aarch64/functions/tme/DiscardTransactionalWrites

```
// DiscardTransactionalWrites()
// =====
// Discards all transactional writes to memory and reset the transactional
// read and write sets.

DiscardTransactionalWrites();
```

## Library pseudocode for aarch64/functions/tme/FailTransaction

```
// FailTransaction()
// =====

FailTransaction(TMFailure cause, boolean retry)
    FailTransaction(cause, retry, FALSE, Zeros(15));
    return;

// FailTransaction()
// =====
// Exits Transactional state and discards transactional updates to registers
// and memory.

FailTransaction(TMFailure cause, boolean retry, boolean interrupt, bits(15) reason)
    assert !retry || !interrupt;

    if HaveBRBExt() && BranchRecordAllowed(PSTATE.EL) then BRBFCE_EL1.LASTFAILED = '1';

    DiscardTransactionalWrites();
    // For trivial implementation no transaction checkpoint was taken
    if cause != TMFailure_TRIVIAL then
        RestoreTransactionCheckpoint();
    ClearExclusiveLocal(ProcessorID());

    bits(64) result = Zeros(64);

    result<23> = if interrupt then '1' else '0';
    result<15> = if retry && !interrupt then '1' else '0';
    case cause of
        when TMFailure_TRIVIAL result<24> = '1';
        when TMFailure_DBG      result<22> = '1';
        when TMFailure_NEST     result<21> = '1';
        when TMFailure_SIZE     result<20> = '1';
        when TMFailure_ERR      result<19> = '1';
        when TMFailure_IMP      result<18> = '1';
        when TMFailure_MEM      result<17> = '1';
        when TMFailure_CNCL     result<16> = '1'; result<14:0> = reason;

    TSTATE.depth = 0;
    X[TSTATE.Rt, 64] = result;
    boolean branch_conditional = FALSE;
    BranchTo(TSTATE.nPC, BranchType TMFAIL, branch_conditional);
    EndOfInstruction();
    return;
```

## Library pseudocode for aarch64/functions/tme/IsTMEEnabled

```
// IsTMEEnabled()
// =====
// Returns TRUE if access to TME instruction is enabled, FALSE otherwise.

boolean IsTMEEnabled()
    if PSTATE.EL IN {EL0, EL1, EL2} && HaveEL(EL3) then
        if SCR_EL3.TME == '0' then
            return FALSE;
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        if HCR_EL2.TME == '0' then
            return FALSE;
    return TRUE;
```

## Library pseudocode for aarch64/functions/tme/MemHasTransactionalAccess

```
// MemHasTransactionalAccess()
// =====
// Function checks if transactional accesses are not supported for an address
// range or memory type.

boolean MemHasTransactionalAccess(MemoryAttributes memattrs)
    if ((memattrs.shareability == Shareability\_ISH ||
        memattrs.shareability == Shareability\_OSH) &&
        memattrs.memtype == MemType\_Normal &&
        memattrs.inner.attrs == MemAttr\_WB &&
        memattrs.inner.hints == MemHint\_RWA &&
        memattrs.inner.transient == FALSE &&
        memattrs.outer.hints == MemHint\_RWA &&
        memattrs.outer.attrs == MemAttr\_WB &&
        memattrs.outer.transient == FALSE) then
        return TRUE;
    else
        return boolean IMPLEMENTATION_DEFINED "Memory Region does not support Transactional access";
```

## Library pseudocode for aarch64/functions/tme/RestoreTransactionCheckpoint

```
// RestoreTransactionCheckpoint()
// =====
// Restores part of the PE registers from the transaction checkpoint.

RestoreTransactionCheckpoint()
    SP[] = TSTATE.SP;
    ICC_PMR_EL1 = TSTATE.ICC_PMR_EL1;
    PSTATE.<N,Z,C,V> = TSTATE.nzcv;
    PSTATE.<D,A,I,F> = TSTATE.<D,A,I,F>;

    for n = 0 to 30
        X[n, 64] = TSTATE.X[n];

    if IsFPEEnabled(PSTATE.EL) then
        if IsSVEEnabled(PSTATE.EL) then
            constant integer VL = CurrentVL;
            constant integer PL = VL DIV 8;
            for n = 0 to 31
                Z[n, VL] = TSTATE.Z[n]<VL-1:0>;
            for n = 0 to 15
                P[n, PL] = TSTATE.P[n]<PL-1:0>;
                FFR[PL] = TSTATE.FFR<PL-1:0>;
        else
            for n = 0 to 31
                V[n, 128] = TSTATE.Z[n]<127:0>;
            FPCR = TSTATE.FPCR;
            FPSR = TSTATE.FPSR;

    if HaveGCS() then
        case PSTATE.EL of
            when EL0 GCSPR_EL0 = TSTATE.GCSPR_ELx;
            when EL1 GCSPR_EL1 = TSTATE.GCSPR_ELx;
            when EL2 GCSPR_EL2 = TSTATE.GCSPR_ELx;
            when EL3 GCSPR_EL3 = TSTATE.GCSPR_ELx;

    return;
```

## Library pseudocode for aarch64/functions/tme/StartTrackingTransactionalReadsWrites

```
// StartTrackingTransactionalReadsWrites()
// =====
// Starts tracking transactional reads and writes to memory.

StartTrackingTransactionalReadsWrites();
```

## Library pseudocode for aarch64/functions/tme/TMFailure

```
// TMFailure
// =====
// Transactional failure causes

enumeration TMFailure {
    TMFailure_CNCL,    // Executed a TCANCEL instruction
    TMFailure_DBG,     // A debug event was generated
    TMFailure_ERR,     // A non-permissible operation was attempted
    TMFailure_NEST,    // The maximum transactional nesting level was exceeded
    TMFailure_SIZE,    // The transactional read or write set limit was exceeded
    TMFailure_MEM,     // A transactional conflict occurred
    TMFailure_TRIVIAL, // Only a TRIVIAL version of TM is available
    TMFailure_IMP      // Any other failure cause
};
```

## Library pseudocode for aarch64/functions/tme/TMState

```
// TMState
// =====
// Transactional execution state bits.
// There is no significance to the field order.

type TMState is (
    integer    depth,           // Transaction nesting depth
    integer    Rt,              // TSTART destination register
    bits(64)   npc,             // Fallback instruction address
    array[0..30] of bits(64) X, // General purpose registers
    array[0..31] of bits(MAX_VL) Z, // Vector registers
    array[0..15] of bits(MAX_PL) P, // Predicate registers
    bits(MAX_PL) FFR,           // First Fault Register
    bits(64)   SP,              // Stack Pointer at current EL
    bits(64)   FPCR,            // Floating-point Control Register
    bits(64)   FPSR,            // Floating-point Status Register
    bits(64)   ICC_PMR_EL1,     // Interrupt Controller Interrupt Priority Mask Register
    bits(64)   GCSPR_ELx,       // GCS pointer for current EL
    bits(4)    nzcv,            // Condition flags
    bits(1)    D,               // Debug mask bit
    bits(1)    A,               // SError interrupt mask bit
    bits(1)    I,               // IRQ mask bit
    bits(1)    F                // FIQ mask bit
)
```

## Library pseudocode for aarch64/functions/tme/TSTATE

```
TMState TSTATE;
```

## Library pseudocode for aarch64/functions/tme/TakeTransactionCheckpoint

```
// TakeTransactionCheckpoint()
// =====
// Captures part of the PE registers into the transaction checkpoint.

TakeTransactionCheckpoint()
    TSTATE.SP          = SP[];
    TSTATE.ICC_PMR_EL1 = ICC_PMR_EL1;
    TSTATE.nzcv        = PSTATE.<N,Z,C,V>;
    TSTATE.<D,A,I,F>    = PSTATE.<D,A,I,F>;

    for n = 0 to 30
        TSTATE.X[n] = X[n, 64];

    if IsFPEEnabled(PSTATE.EL) then
        if IsSVEEnabled(PSTATE.EL) then
            constant integer VL = CurrentVL;
            constant integer PL = VL DIV 8;
            for n = 0 to 31
                TSTATE.Z[n]<VL-1:0> = Z[n, VL];
            for n = 0 to 15
                TSTATE.P[n]<PL-1:0> = P[n, PL];
            TSTATE.FFR<PL-1:0> = FFR[PL];
        else
            for n = 0 to 31
                TSTATE.Z[n]<127:0> = V[n, 128];
            TSTATE.FPCR = FPCR;
            TSTATE.FPSR = FPSR;

    if HaveGCS() then
        case PSTATE.EL of
            when EL0 TSTATE.GCSPR_ELx = GCSPR_EL0;
            when EL1 TSTATE.GCSPR_ELx = GCSPR_EL1;
            when EL2 TSTATE.GCSPR_ELx = GCSPR_EL2;
            when EL3 TSTATE.GCSPR_ELx = GCSPR_EL3;

    return;
```

## Library pseudocode for aarch64/functions/tme/TransactionStartTrap

```
// TransactionStartTrap()
// =====
// Traps the execution of TSTART instruction.

TransactionStartTrap(integer dreg)
    bits(2) targetEL;
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception TSTARTAccessTrap);
    except.syndrome<9:5> = dreg<4:0>;

    if UInt(PSTATE.EL) > UInt(EL1) then
        targetEL = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        targetEL = EL2;
    else
        targetEL = EL1;
    AArch64.TakeException(targetEL, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/functions/vbitop/VBitOp

```
// VBitOp
// =====
// Vector bit select instruction types.

enumeration VBitOp    {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};
```



## Library pseudocode for aarch64/translation/attrs/AArch64.MAIRAttr

```
// AArch64.MAIRAttr()
// =====
// Retrieve the memory attribute encoding indexed in the given MAIR

bits(8) AArch64.MAIRAttr(integer index, MAIRType mair2, MAIRType mair)
    bit_index = 8 * index;
    assert (index < 8 || (HaveAIEExt() && (index < 16)));
    if (index > 7) then
        bit_index = bit_index - 64; // Read from LSB at MAIR2
        return mair2<bit_index+7:bit_index>;
    else
        return mair<bit_index+7:bit_index>;
```

## Library pseudocode for aarch64/translation/debug/AArch64.CheckBreakpoint

```
// AArch64.CheckBreakpoint()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch64.CheckBreakpoint(FaultRecord fault_in, bits(64) vaddress,
                                     AccessDescriptor accdesc, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert (UsingAArch32() && size IN {2,4}) || size == 4;

    FaultRecord fault = fault_in;
    boolean match = FALSE;
    boolean mismatch = TRUE; // Default assumption that all mismatches are outside
                             // the range of all address match breakpoints
    boolean mismatch_bp = FALSE; // Has a breakpoint been configured for a mismatch

    for i = 0 to NumBreakpointsImplemented() - 1
        (match_i, is_mismatch_i) = AArch64.BreakpointMatch(i, vaddress, accdesc, size);
        if is_mismatch_i then
            mismatch_bp = TRUE;
            mismatch = mismatch && !match_i;
        else
            match = match || match_i;

    if match || (mismatch && mismatch_bp) then
        fault.statuscode = Fault_Debug;
        if HaltOnBreakpointOrWatchpoint() then
            reason = DebugHalt_Breakpoint;
            Halt(reason);

    return fault;
```

## Library pseudocode for aarch64/translation/debug/AArch64.CheckDebug

```
// AArch64.CheckDebug()
// =====
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccessDescriptor accdesc, integer size)

    FaultRecord fault = NoFault(accdesc);
    boolean generate_exception;

    boolean d_side = (IsDataAccess(accdesc.acctype) || accdesc.acctype == AccessType\_DC);
    boolean i_side = (accdesc.acctype == AccessType\_IFETCH);
    if accdesc.acctype == AccessType\_NV2 then
        mask = '0';
        ss = CurrentSecurityState();
        generate_exception = (AArch64.GenerateDebugExceptionsFrom(EL2, ss, mask) &&
                               MDSCR_EL1.MDE == '1');
    else
        generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
    halt = HaltOnBreakpointOrWatchpoint();

    if generate_exception || halt then
        if d_side then
            fault = AArch64.CheckWatchpoint(fault, vaddress, accdesc, size);
        elseif i_side then
            fault = AArch64.CheckBreakpoint(fault, vaddress, accdesc, size);

    return fault;
```

## Library pseudocode for aarch64/translation/debug/AArch64.CheckWatchpoint

```
// AArch64.CheckWatchpoint()
// =====
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch64.CheckWatchpoint(FaultRecord fault_in, bits(64) vaddress_in,
                                     AccessDescriptor accdesc, integer size_in)
    assert !ELUsingAArch32(S1TranslationRegime());
    FaultRecord fault          = fault_in;
    bits(64) vaddress          = vaddress_in;
    integer size                = size_in;
    boolean rounded_match      = FALSE;
    bits(64) original_vaddress = vaddress;
    integer original_size       = size;

    if accdesc.acctype == AccessType\_DC then
        if accdesc.cacheop != CacheOp\_Invalidate then
            return fault;
    elseif !IsDataAccess(accdesc.acctype) then
        return fault;

    // In case of set of contiguous memory accesses each call to this function is such that:
    // - the lowest accessed address is rounded down to the nearest multiple of 16 bytes
    // - the highest accessed address is rounded up to the nearest multiple of 16 bytes
    // Since the WPF field is set if the implementation does rounding, regardless of true or
    // false match, it would be acceptable to return TRUE for either/both of the first and last
    // access.
    if IsSVEAccess(accdesc) || IsSMEAAccess(accdesc) then
        integer upper_vaddress = UInt(original_vaddress) + original_size;
        if ConstrainUnpredictableBool(Unpredictable\_16BYTEROUNDEDDOWNACCESS) then
            vaddress = Align(vaddress, 16);
            rounded_match = TRUE;
        if ConstrainUnpredictableBool(Unpredictable\_16BYTEROUNDEDUPACCESS) then
            upper_vaddress = Align(upper_vaddress + 15, 16) ;
            rounded_match = TRUE;
        size = upper_vaddress - UInt(vaddress);

    for i = 0 to NumWatchpointsImplemented() - 1
        if AArch64.WatchpointMatch(i, vaddress, size, accdesc) then
            fault.maybe_false_match = rounded_match;
            fault.watchpt_num = i;
            fault.statuscode = Fault\_Debug;
            if DBGWCR_EL1[i].LSC<0> == '1' && accdesc.read then
                fault.write = FALSE;
            elseif DBGWCR_EL1[i].LSC<1> == '1' && accdesc.write then
                fault.write = TRUE;
        if (fault.statuscode == Fault\_Debug && HaltOnBreakpointOrWatchpoint() &&
            !accdesc.nonfault && !(accdesc.firstfault && !accdesc.first)) then
            reason = DebugHalt\_Watchpoint;
            EDWAR = vaddress;
            is_async = FALSE;
            Halt(reason, is_async, fault);
    return fault;
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.IASize

```
// AArch64.IASize()
// =====
// Retrieve the number of bits containing the input address

integer AArch64.IASize(bits(6) txsz)
    return 64 - UInt(txsz);
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.LeafBase

```
// AArch64.LeafBase()
// =====
// Extract the address embedded in a block and page descriptor pointing to the
// base of a memory block

bits(56) AArch64.LeafBase(bits(N) descriptor, bit d128, bit ds,
                           TGx tgx, integer level)
    bits(56) leafbase = Zeros(56);

    granulebits = TGxGranuleBits(tgx);
    descsize2 = if d128 == '1' then 4 else 3;
    stride = granulebits - descsize2;
    leafsize = granulebits + stride * (FINAL_LEVEL - level);

    leafbase<47:0> = Align(descriptor<47:0>, 1 << leafsize);

    if Have56BitPAExt() && d128 == '1' then
        leafbase<55:48> = descriptor<55:48>;
        return leafbase;
    if Have52BitPAExt() && tgx == TGx_64KB then
        leafbase<51:48> = descriptor<15:12>;
    elsif ds == '1' then
        leafbase<51:48> = descriptor<9:8>:descriptor<49:48>;

    return leafbase;
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.NextTableBase

```
// AArch64.NextTableBase()
// =====
// Extract the address embedded in a table descriptor pointing to the base of
// the next level table of descriptors

bits(56) AArch64.NextTableBase(bits(N) descriptor, bit d128, bits(2) skl, bit ds, TGx tgx)
    bits(56) tablebase = Zeros(56);
    integer granulebits = TGxGranuleBits(tgx);
    integer tablesize;

    if d128 == '1' then
        integer descsize2 = 4;
        integer stride = granulebits - descsize2;
        tablesize = stride*(1 + UInt(skl)) + descsize2;
    else
        tablesize = granulebits;

    case tgx of
        when TGx_4KB tablebase<47:12> = descriptor<47:12>;
        when TGx_16KB tablebase<47:14> = descriptor<47:14>;
        when TGx_64KB tablebase<47:16> = descriptor<47:16>;

    tablebase = Align(tablebase, 1 << tablesize);

    if Have56BitPAExt() && d128 == '1' then
        tablebase<55:48> = descriptor<55:48>;
        return tablebase;
    if Have52BitPAExt() && tgx == TGx_64KB then
        tablebase<51:48> = descriptor<15:12>;
        return tablebase;
    if ds == '1' then
        tablebase<51:48> = descriptor<9:8>:descriptor<49:48>;
        return tablebase;
    return tablebase;
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.PhysicalAddressSize

```
// AArch64.PhysicalAddressSize()
// =====
// Retrieve the number of bits bounding the physical address

integer AArch64.PhysicalAddressSize(bit d128, bits(3) encoded_ps, TGx tgx)
    integer ps;
    integer max_ps;

    case encoded_ps of
        when '000' ps = 32;
        when '001' ps = 36;
        when '010' ps = 40;
        when '011' ps = 42;
        when '100' ps = 44;
        when '101' ps = 48;
        when '110' ps = 52;
        when '111' ps = 56;
    if !Have56BitPAExt() || d128 == '0' then
        if tgx != TGx\_64KB && !Have52BitIPAAndPASpaceExt() then
            max_ps = Min(48, AArch64.PAMax());
        elseif !Have52BitPAExt() then
            max_ps = Min(48, AArch64.PAMax());
        else
            max_ps = Min(52, AArch64.PAMax());
    else
        max_ps = AArch64.PAMax();

    return Min(ps, max_ps);
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.S1SLTTEEntryAddress

```
// AArch64.S1SLTTEEntryAddress()
// =====
// Compute the first stage 1 translation table descriptor address within the
// table pointed to by the base at the start level

FullAddress AArch64.S1SLTTEEntryAddress(integer level, S1TTWParams walkparams,
                                         bits(64) ia, FullAddress tablebase)

    // Input Address size
    iasize      = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    descsize2   = if walkparams.d128 == '1' then 4 else 3;
    stride      = granulebits - descsize2;
    levels      = FINAL\_LEVEL - level;

    bits(56) index;
    lsb  = levels*stride + granulebits;
    msb  = iasize - 1;
    index = ZeroExtend(ia<msb:lsb>:Zeros(descsize2), 56);

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.S1StartLevel

```
// AArch64.S1StartLevel()
// =====
// Compute the initial lookup level when performing a stage 1 translation
// table walk

integer AArch64.S1StartLevel(S1TTWParams walkparams)
    // Input Address size
    iasize      = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    descsize2   = if walkparams.d128 == '1' then 4 else 3;
    stride      = granulebits - descsize2;
    slstartlevel = FINAL\_LEVEL - ((iasize-1) - granulebits) DIV stride;
    if walkparams.d128 == '1' then
        slstartlevel = slstartlevel + UInt(walkparams.skl);
    return slstartlevel;
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.S1TTBaseAddress

```
// AArch64.S1TTBaseAddress()
// =====
// Retrieve the PA/IPA pointing to the base of the initial translation table of stage 1

bits(56) AArch64.S1TTBaseAddress(S1TTWParams walkparams, Regime regime, bits(N) ttbr)
    bits(56) tablebase = Zeros(56);

    // Input Address size
    iasize      = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    descsize2   = if walkparams.d128 == '1' then 4 else 3;
    stride      = granulebits - descsize2;
    startlevel  = AArch64.S1StartLevel(walkparams);
    levels      = FINAL\_LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table in bytes
    tsize = (iasize - (levels*stride + granulebits)) + descsize2;

    if Have56BitPAExt() && walkparams.d128 == '1' then
        tsize = Max(tsize, 5);
        if regime == Regime\_EL3 then
            tablebase<55:5> = ttbr<55:5>;
        else
            tablebase<55:5> = ttbr<87:80>:ttbr<47:5>;
    elseif ((Have52BitPAExt() && walkparams.tgx == TGx\_64KB && walkparams.ps == '110') ||
            (walkparams.ds == '1')) then
        tsize = Max(tsize, 6);
        tablebase<51:6> = ttbr<5:2>:ttbr<47:6>;
    else
        tablebase<47:1> = ttbr<47:1>;
    tablebase = Align(tablebase, 1 << tsize);
    return tablebase;
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.S2SLTTEntryAddress

```
// AArch64.S2SLTTEntryAddress()
// =====
// Compute the first stage 2 translation table descriptor address within the
// table pointed to by the base at the start level

FullAddress AArch64.S2SLTTEntryAddress(S2TTWParams walkparams, bits(56) ipa,
                                       FullAddress tablebase)

    startlevel    = AArch64.S2StartLevel(walkparams);
    iasize        = AArch64.IASize(walkparams.txsz);
    granulebits   = TGxGranuleBits(walkparams.tgx);
    descsize2     = if walkparams.dl28 == '1' then 4 else 3;
    stride        = granulebits - descsize2;
    levels        = FINAL\_LEVEL - startlevel;

    bits(56) index;
    integer lsb;
    integer msb;
    lsb  = levels*stride + granulebits;
    msb  = iasize - 1;
    index = ZeroExtend(ipa<msb:lsb>:Zeros(descsize2), 56);

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.S2StartLevel

```
// AArch64.S2StartLevel()
// =====
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch64.S2StartLevel(S2TTWParams walkparams)
    if walkparams.dl28 == '1' then
        iasize        = AArch64.IASize(walkparams.txsz);
        granulebits   = TGxGranuleBits(walkparams.tgx);
        descsize2     = 4;
        stride        = granulebits - descsize2;
        s2startlevel  = FINAL\_LEVEL - (((iasize-1) - granulebits) DIV stride);
        s2startlevel  = s2startlevel + UInt(walkparams.sk1);

        return s2startlevel;

    case walkparams.tgx of
        when TGx\_4KB
            case walkparams.sl2:walkparams.sl0 of
                when '000' return 2;
                when '001' return 1;
                when '010' return 0;
                when '011' return 3;
                when '100' return -1;
        when TGx\_16KB
            case walkparams.sl0 of
                when '00' return 3;
                when '01' return 2;
                when '10' return 1;
                when '11' return 0;
        when TGx\_64KB
            case walkparams.sl0 of
                when '00' return 3;
                when '01' return 2;
                when '10' return 1;
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.S2TTBaseAddress

```
// AArch64.S2TTBaseAddress()
// =====
// Retrieve the PA/IPA pointing to the base of the initial translation table of stage 2

bits(56) AArch64.S2TTBaseAddress(S2TTWParams walkparams, PASpace paspace, bits(N) ttbr)
    bits(56) tablebase = Zeros(56);

    // Input Address size
    iasize      = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    descsizeLog2 = if walkparams.d128 == '1' then 4 else 3;
    stride      = granulebits - descsizeLog2;
    startlevel  = AArch64.S2StartLevel(walkparams);
    levels      = FINAL_LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table in bytes
    tsize = (iasize - (levels*stride + granulebits)) + descsizeLog2;

    if Have56BitPAExt() && walkparams.d128 == '1' then
        tsize = Max(tsize, 5);
        if paspace == PAS_Secure then
            tablebase<55:5> = ttbr<55:5>;
        else
            tablebase<55:5> = ttbr<87:80>:ttbr<47:5>;
    elseif ((Have52BitPAExt() && walkparams.tgx == TGx_64KB && walkparams.ps == '110') ||
            (walkparams.ds == '1')) then
        tsize = Max(tsize, 6);
        tablebase<51:6> = ttbr<5:2>:ttbr<47:6>;
    else
        tablebase<47:1> = ttbr<47:1>;
    tablebase = Align(tablebase, 1 << tsize);
    return tablebase;
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.TTEntryAddress

```
// AArch64.TTEntryAddress()
// =====
// Compute translation table descriptor address within the table pointed to by
// the table base

FullAddress AArch64.TTEntryAddress(integer level, bit d128, bits(2) skl, TGx tgx, bits(6) txsz,
    bits(64) ia, FullAddress tablebase)

    // Input Address size
    iasize      = AArch64.IASize(txsz);
    granulebits = TGxGranuleBits(tgx);
    descsizeLog2 = if d128 == '1' then 4 else 3;
    stride      = granulebits - descsizeLog2;
    levels      = FINAL_LEVEL - level;

    bits(56) index;
    integer lsb;
    integer msb;

    lsb = levels*stride + granulebits;
    if d128 == '1' then
        msb = (lsb + stride*(1 + UInt(skl))) - 1;
    else
        msb = (lsb + stride) - 1;
    index = ZeroExtend(ia<msb:lsb>:Zeros(descsizeLog2), 56);

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
```



### Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.AddrTop

```
// AArch64.AddrTop()
// =====
// Get the top bit position of the virtual address.
// Bits above are not accounted as part of the translation process.

integer AArch64.AddrTop(bit tbid, AccessType acctype, bit tbi)
    if tbid == '1' && acctype == AccessType\_IFETCH then
        return 63;

    if tbi == '1' then
        return 55;
    else
        return 63;
```

### Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.ContiguousBitFaults

```
// AArch64.ContiguousBitFaults()
// =====
// If contiguous bit is set, returns whether the translation size exceeds the
// input address size and if the implementation generates a fault

boolean AArch64.ContiguousBitFaults(bit d128, bits(6) txsz, TGx tgx, integer level)
    // Input Address size
    iasize = AArch64.IASize(txsz);
    // Translation size
    tsize = TranslationSize(d128, tgx, level) + ContiguousSize(d128, tgx, level);

    return (tsize > iasize &&
        boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit");
```

### Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.IPAIsOutOfRange

```
// AArch64.IPAIsOutOfRange()
// =====
// Check bits not resolved by translation are ZERO

boolean AArch64.IPAIsOutOfRange(bits(56) ipa, S2TTWParams walkparams)
    //Input Address size
    iasize = AArch64.IASize(walkparams.txsz);

    if iasize < 56 then
        return !IsZero(ipa<55:iasize>);
    else
        return FALSE;
```

### Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.OAOutOfRange

```
// AArch64.OAOutOfRange()
// =====
// Returns whether output address is expressed in the configured size number of bits

boolean AArch64.OAOutOfRange(bits(56) address, bit d128, bits(3) ps, TGx tgx)
    // Output Address size
    oasize = AArch64.PhysicalAddressSize(d128, ps, tgx);

    if oasize < 56 then
        return !IsZero(address<55:oasize>);
    else
        return FALSE;
```



```

// AArch64.S1CheckPermissions()
// =====
// Checks whether stage 1 access violates permissions of target memory
// and returns a fault record

FaultRecord AArch64.S1CheckPermissions(FaultRecord fault_in, Regime regime, TTWState walkstate,
                                         S1TTWParams walkparams, AccessDescriptor accdesc)

FaultRecord fault = fault_in;
Permissions permissions = walkstate.permissions;
S1AccessControls slperms;

slperms = AArch64.S1ComputePermissions(regime, walkstate, walkparams, accdesc);

if accdesc.acctype == AccessType\_IFETCH then
    if slperms.overlay && slperms.ox == '0' then
        fault.statuscode = Fault\_Permission;
        fault.overlay = TRUE;
    elseif (walkstate.memattrs.memtype == MemType\_Device &&
            ConstrainUnpredictable(Unpredictable\_INSTRDEVICE) == Constraint\_FAULT) then
        fault.statuscode = Fault\_Permission;
    elseif slperms.x == '0' then
        fault.statuscode = Fault\_Permission;
elseif accdesc.acctype == AccessType\_DC then
    if accdesc.cacheop == CacheOp\_Invalidate then
        if slperms.overlay && slperms.ow == '0' then
            fault.statuscode = Fault\_Permission;
            fault.overlay = TRUE;
        elseif slperms.w == '0' then
            fault.statuscode = Fault\_Permission;
        // DC from privileged context which clean cannot generate a Permission fault
    elseif accdesc.el == EL0 then
        if slperms.overlay && slperms.or == '0' then
            fault.statuscode = Fault\_Permission;
            fault.overlay = TRUE;
        elseif (walkparams.cmow == '1' &&
                accdesc.opscope == CacheOpScope\_PoC &&
                accdesc.cacheop == CacheOp\_CleanInvalidate &&
                slperms.overlay && slperms.ow == '0') then
            fault.statuscode = Fault\_Permission;
            fault.overlay = TRUE;
        elseif slperms.r == '0' then
            fault.statuscode = Fault\_Permission;
        elseif (walkparams.cmow == '1' &&
                accdesc.opscope == CacheOpScope\_PoC &&
                accdesc.cacheop == CacheOp\_CleanInvalidate &&
                slperms.w == '0') then
            fault.statuscode = Fault\_Permission;
elseif accdesc.acctype == AccessType\_IC then
    // IC from privileged context cannot generate Permission fault
    if accdesc.el == EL0 then
        if (slperms.overlay && slperms.or == '0' &&
            boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
            fault.statuscode = Fault\_Permission;
            fault.overlay = TRUE;
        elseif walkparams.cmow == '1' && slperms.overlay && slperms.ow == '0' then
            fault.statuscode = Fault\_Permission;
            fault.overlay = TRUE;
        elseif (slperms.r == '0' &&
            boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
            fault.statuscode = Fault\_Permission;
        elseif walkparams.cmow == '1' && slperms.w == '0' then
            fault.statuscode = Fault\_Permission;
    elseif HaveGCS() && accdesc.acctype == AccessType\_GCS then
        if slperms.gcs == '0' then
            fault.statuscode = Fault\_Permission;
        elseif accdesc.write && walkparams.<ha,hd> != '11' && permissions.ndirty == '1' then
            fault.statuscode = Fault\_Permission;
            fault.dirtybit = TRUE;
            fault.write = TRUE;
        elseif accdesc.read && slperms.overlay && slperms.or == '0' then

```

```

    fault.statuscode = Fault\_Permission;
    fault.overlay    = TRUE;
    fault.write      = FALSE;
elseif accdesc.write && slperms.overlay && slperms.ow == '0' then
    fault.statuscode = Fault\_Permission;
    fault.overlay    = TRUE;
    fault.write      = TRUE;
elseif accdesc.read && slperms.r == '0' then
    fault.statuscode = Fault\_Permission;
    fault.write      = FALSE;
elseif accdesc.write && slperms.w == '0' then
    fault.statuscode = Fault\_Permission;
    fault.write      = TRUE;
elseif (accdesc.write && accdesc.tagaccess &&
        walkstate.memattrs.tags == MemTag\_CanonicallyTagged) then
    fault.statuscode = Fault\_Permission;
    fault.write      = TRUE;
    fault.sltagnodata = TRUE;
elseif (accdesc.write && !(walkparams.<ha,hd> == '11') && walkparams.pie == '1' &&
        permissions.ndirty == '1') then
    fault.statuscode = Fault\_Permission;
    fault.dirtybit   = TRUE;
    fault.write      = TRUE;

return fault;

```

### Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S1ComputePermissions

```

// AArch64.S1ComputePermissions()
// =====
// Computes the overall stage 1 permissions

S1AccessControls AArch64.S1ComputePermissions(Regime regime, TTWState walkstate,
                                              S1TTWParams walkparams, AccessDescriptor accdesc)
Permissions permissions = walkstate.permissions;
S1AccessControls slperms;

if walkparams.pie == '1' then
    slperms = AArch64.S1IndirectBasePermissions(regime, walkstate, walkparams, accdesc);
else
    slperms = AArch64.S1DirectBasePermissions(regime, walkstate, walkparams, accdesc);

if accdesc.el == EL0 && !AArch64.S1E0POEnabled(regime, walkparams.nv1) then
    slperms.overlay = FALSE;
elseif accdesc.el != EL0 && !AArch64.S1POEnabled(regime) then
    slperms.overlay = FALSE;

if slperms.overlay then
    sloverlay_perms = AArch64.S1OverlayPermissions(regime, walkstate, accdesc);
    slperms.or = sloverlay_perms.or;
    slperms.ow = sloverlay_perms.ow;
    slperms.ox = sloverlay_perms.ox;

// If wxn is set, overlay execute permissions is set to 0
if slperms.overlay && slperms.wxn == '1' && slperms.ox == '1' then
    slperms.ow = '0';
elseif slperms.wxn == '1' then
    slperms.x = '0';

return slperms;

```



```

// AArch64.S1DirectBasePermissions()
// =====
// Computes the stage 1 direct base permissions

S1AccessControls AArch64.S1DirectBasePermissions(Regime regime, TTWState walkstate,
S1TTWParams walkparams, AccessDescriptor accdesc)

bit r, w, x;
bit pr, pw, px;
bit ur, uw, ux;
Permissions permissions = walkstate.permissions;
S1AccessControls slperms;

if HasUnprivileged(regime) then
    // Apply leaf permissions
    case permissions.ap<2:1> of
        when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // Privileged access
        when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // No effect
        when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // Read-only, privileged access
        when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // Read-only

    // Apply hierarchical permissions
    case permissions.ap_table of
        when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
        when '01' (pr,pw,ur,uw) = ( pr, pw, '0','0'); // Privileged access
        when '10' (pr,pw,ur,uw) = ( pr, '0', ur, '0'); // Read-only
        when '11' (pr,pw,ur,uw) = ( pr, '0', '0','0'); // Read-only, privileged access

    // Locations writable by unprivileged cannot be executed by privileged
    px = NOT(permissions.pxn OR permissions.pxn_table OR uw);
    ux = NOT(permissions.uxn OR permissions.uxn_table);

    if HavePANExt() && accdesc.pan && !(regime == Regime\_EL10 && walkparams.nv1 == '1') then
        bit pan;
        if (boolean IMPLEMENTATION_DEFINED "SCR_EL3.SIF affects EPAN" &&
            accdesc.ss == SS\_Secure &&
            walkstate.baseaddress.paspace == PAS\_NonSecure &&
            walkparams.sif == '1') then
            ux = '0';

        if (boolean IMPLEMENTATION_DEFINED "Realm EL2&0 regime affects EPAN" &&
            accdesc.ss == SS\_Realm && regime == Regime\_EL20 &&
            walkstate.baseaddress.paspace != PAS\_Realm) then
            ux = '0';

        pan = PSTATE.PAN AND (ur OR uw OR (walkparams.epan AND ux));
        pr = pr AND NOT(pan);
        pw = pw AND NOT(pan);

    else
        // Apply leaf permissions
        case permissions.ap<2> of
            when '0' (pr,pw) = ('1','1'); // No effect
            when '1' (pr,pw) = ('1','0'); // Read-only

        // Apply hierarchical permissions
        case permissions.ap_table<1> of
            when '0' (pr,pw) = ( pr, pw); // No effect
            when '1' (pr,pw) = ( pr, '0'); // Read-only

        px = NOT(permissions.xn OR permissions.xn_table);

    (r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

    // Compute WXN value
    wxn = walkparams.wxn AND w AND x;

    // Prevent execution from Non-secure space by PE in secure state if SIF is set
    if accdesc.ss == SS\_Secure && walkstate.baseaddress.paspace == PAS\_NonSecure then
        x = x AND NOT(walkparams.sif);
    // Prevent execution from non-Root space by Root

```

```

if accdesc.ss == SS\_Root && walkstate.baseaddress.paspace != PAS\_Root then
    x = '0';
// Prevent execution from non-Realm space by Realm EL2 and Realm EL2&0
if (accdesc.ss == SS\_Realm && regime IN {Regime\_EL2, Regime\_EL20} &&
    walkstate.baseaddress.paspace != PAS\_Realm) then
    x = '0';

slperms.r    = r;
slperms.w    = w;
slperms.x    = x;
slperms.gcs  = '0';
slperms.wxn  = wxn;
slperms.overlay = TRUE;

return slperms;

```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S1HasAlignmentFault

```

// AArch64.S1HasAlignmentFault()
// =====
// Returns whether stage 1 output fails alignment requirement on data accesses
// to Device memory

boolean AArch64.S1HasAlignmentFault(AccessDescriptor accdesc, boolean aligned,
                                     bit ntlsmd, MemoryAttributes memattrs)
    if accdesc.acctype == AccessType\_IFETCH then
        return FALSE;
    elsif HaveMTEExt() && accdesc.tagaccess && accdesc.write then
        return (memattrs.memtype == MemType\_Device &&
            ConstrainUnpredictable(Unpredictable\_DEVICETAGSTORE) == Constraint\_FAULT);
    elsif accdesc.a32lsmd && ntlsmd == '0' then
        return memattrs.memtype == MemType\_Device && memattrs.device != DeviceType\_GRE;
    elsif accdesc.acctype == AccessType\_DCZero then
        return memattrs.memtype == MemType\_Device;
    else
        return memattrs.memtype == MemType\_Device && !aligned;

```





```

// AArch64.S1IndirectBasePermissions()
// =====
// Computes the stage 1 indirect base permissions

S1AccessControls AArch64.S1IndirectBasePermissions(Regime regime, TTWState walkstate,
S1TTWParams walkparams,
AccessDescriptor accdesc)

bit r, w, x, gcs, wxn, overlay;
bit pr, pw, px, pgcs, pwxn, p_overlay;
bit ur, uw, ux, ugcs, uwxn, u_overlay;
Permissions permissions = walkstate.permissions;
S1AccessControls slperms;

// Apply privileged indirect permissions
case permissions.ppi of
    when '0000' (pr,pw,px,pgcs) = ('0','0','0','0'); // No access
    when '0001' (pr,pw,px,pgcs) = ('1','0','0','0'); // Privileged read
    when '0010' (pr,pw,px,pgcs) = ('0','0','1','0'); // Privileged execute
    when '0011' (pr,pw,px,pgcs) = ('1','0','1','0'); // Privileged read and execute
    when '0100' (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
    when '0101' (pr,pw,px,pgcs) = ('1','1','0','0'); // Privileged read and write
    when '0110' (pr,pw,px,pgcs) = ('1','1','1','0'); // Privileged read, write and execute
    when '0111' (pr,pw,px,pgcs) = ('1','1','1','0'); // Privileged read, write and execute
    when '1000' (pr,pw,px,pgcs) = ('1','0','0','0'); // Privileged read
    when '1001' (pr,pw,px,pgcs) = ('1','0','0','1'); // Privileged read and gcs
    when '1010' (pr,pw,px,pgcs) = ('1','0','1','0'); // Privileged read and execute
    when '1011' (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
    when '1100' (pr,pw,px,pgcs) = ('1','1','0','0'); // Privileged read and write
    when '1101' (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
    when '1110' (pr,pw,px,pgcs) = ('1','1','1','0'); // Privileged read, write and execute
    when '1111' (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved

p_overlay = NOT(permissions.ppi[3]);
pwxn = if permissions.ppi == '0110' then '1' else '0';

if HasUnprivileged(regime) then
    // Apply unprivileged indirect permissions
    case permissions.upi of
        when '0000' (ur,uw,ux,ugcs) = ('0','0','0','0'); // No access
        when '0001' (ur,uw,ux,ugcs) = ('1','0','0','0'); // Unprivileged read
        when '0010' (ur,uw,ux,ugcs) = ('0','0','1','0'); // Unprivileged execute
        when '0011' (ur,uw,ux,ugcs) = ('1','0','1','0'); // Unprivileged read and execute
        when '0100' (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved
        when '0101' (ur,uw,ux,ugcs) = ('1','1','0','0'); // Unprivileged read and write
        when '0110' (ur,uw,ux,ugcs) = ('1','1','1','0'); // Unprivileged read, write and execute
        when '0111' (ur,uw,ux,ugcs) = ('1','1','1','0'); // Unprivileged read, write and execute
        when '1000' (ur,uw,ux,ugcs) = ('1','0','0','0'); // Unprivileged read
        when '1001' (ur,uw,ux,ugcs) = ('1','0','0','1'); // Unprivileged read and gcs
        when '1010' (ur,uw,ux,ugcs) = ('1','0','1','0'); // Unprivileged read and execute
        when '1011' (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved
        when '1100' (ur,uw,ux,ugcs) = ('1','1','0','0'); // Unprivileged read and write
        when '1101' (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved
        when '1110' (ur,uw,ux,ugcs) = ('1','1','1','0'); // Unprivileged read,write and execute
        when '1111' (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved

    u_overlay = NOT(permissions.upi[3]);
    uwxn = if permissions.upi == '0110' then '1' else '0';

    // If the decoded permissions has either px or pgcs along with either uw or ugcs,
    // then all effective Stage 1 Base Permissions are set to 0
    if ((px == '1' || pgcs == '1') && (uw == '1' || ugcs == '1')) then
        (pr,pw,px,pgcs) = ('0','0','0','0');
        (ur,uw,ux,ugcs) = ('0','0','0','0');

    if HavePANExt() && accdesc.pan && !(regime == Regime\_EL10 && walkparams.nv1 == '1') then
        if PSTATE.PAN == '1' && (permissions.upi != '0000') then
            (pr,pw) = ('0','0');

if accdesc.el == EL0 then

```

```

        (r,w,x,gcs,wxn,overlay) = (ur,uw,ux,ugcs,uwxn,u_overlay);
else
    (r,w,x,gcs,wxn,overlay) = (pr,pw,px,pgcs,pwxn,p_overlay);

// Prevent execution from Non-secure space by PE in secure state if SIF is set
if accdesc.ss == SS\_Secure && walkstate.baseaddress.paspace == PAS\_NonSecure then
    x = x AND NOT(walkparams.sif);
    gcs = '0';
// Prevent execution from non-Root space by Root
if accdesc.ss == SS\_Root && walkstate.baseaddress.paspace != PAS\_Root then
    x = '0';
    gcs = '0';
// Prevent execution from non-Realm space by Realm EL2 and Realm EL2&0
if (accdesc.ss == SS\_Realm && regime IN {Regime\_EL2, Regime\_EL20} &&
    walkstate.baseaddress.paspace != PAS\_Realm) then
    x = '0';
    gcs = '0';

slperms.r      = r;
slperms.w      = w;
slperms.x      = x;
slperms.gcs    = gcs;
slperms.wxn    = wxn;
slperms.overlay = overlay == '1';

return slperms;

```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S1OverlayPermissions

```
// AArch64.S1OverlayPermissions()
// =====
// Computes the stage 1 overlay permissions

S1AccessControls AArch64.S1OverlayPermissions(Regime regime, TTWState walkstate,
                                              AccessDescriptor accdesc)

bit r, w, x;
bit pr, pw, px;
bit ur, uw, ux;
Permissions permissions = walkstate.permissions;
S1AccessControls sloverlay_perms;

S1PORType por = AArch64.S1POR(regime);
integer bit_index = 4 * UInt(permissions.po_index);
bits(4) ppo = por<bit_index+3:bit_index>;

// Apply privileged overlay permissions
case ppo of
  when '0000' (pr,pw,px) = ('0','0','0'); // No access
  when '0001' (pr,pw,px) = ('1','0','0'); // Privileged read
  when '0010' (pr,pw,px) = ('0','0','1'); // Privileged execute
  when '0011' (pr,pw,px) = ('1','0','1'); // Privileged read and execute
  when '0100' (pr,pw,px) = ('0','1','0'); // Privileged write
  when '0101' (pr,pw,px) = ('1','1','0'); // Privileged read and write
  when '0110' (pr,pw,px) = ('0','1','1'); // Privileged write and execute
  when '0111' (pr,pw,px) = ('1','1','1'); // Privileged read, write and execute
  when '1xxx' (pr,pw,px) = ('0','0','0'); // Reserved

if HasUnprivileged(regime) then
  bits(4) upo = POR_EL0<bit_index+3:bit_index>;

  // Apply unprivileged overlay permissions
  case upo of
    when '0000' (ur,uw,ux) = ('0','0','0'); // No access
    when '0001' (ur,uw,ux) = ('1','0','0'); // Unprivileged read
    when '0010' (ur,uw,ux) = ('0','0','1'); // Unprivileged execute
    when '0011' (ur,uw,ux) = ('1','0','1'); // Unprivileged read and execute
    when '0100' (ur,uw,ux) = ('0','1','0'); // Unprivileged write
    when '0101' (ur,uw,ux) = ('1','1','0'); // Unprivileged read and write
    when '0110' (ur,uw,ux) = ('0','1','1'); // Unprivileged write and execute
    when '0111' (ur,uw,ux) = ('1','1','1'); // Unprivileged read, write and execute
    when '1xxx' (ur,uw,ux) = ('0','0','0'); // Reserved

(r,w,x) = if accdesc.el == ELO then (ur,uw,ux) else (pr,pw,px);

sloverlay_perms.or = r;
sloverlay_perms.ow = w;
sloverlay_perms.ox = x;

return sloverlay_perms;
```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S1TxSZFaults

```
// AArch64.S1TxSZFaults()
// =====
// Detect whether configuration of stage 1 TxSZ field generates a fault

boolean AArch64.S1TxSZFaults(Regime regime, S1TTWParams walkparams)
    mintxs = AArch64.S1MinTxSZ(regime, walkparams.d128, walkparams.ds, walkparams.tgx);
    maxtxs = AArch64.MaxTxSZ(walkparams.tgx);

    if UInt(walkparams.txs) < mintxs then
        return (Have52BitVAExt() ||
            boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum");
    if UInt(walkparams.txs) > maxtxs then
        return boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum";

    return FALSE;
```



```

// AArch64.S2CheckPermissions()
// =====
// Verifies memory access with available permissions.

(FaultRecord, boolean) AArch64.S2CheckPermissions(FaultRecord fault_in, TTWState walkstate,
S2TTWParams walkparams, AddressDescriptor ipa,
AccessDescriptor accdesc)

MemType memtype = walkstate.memattrs.memtype;
Permissions permissions = walkstate.permissions;
FaultRecord fault = fault_in;
S2AccessControls s2perms = AArch64.S2ComputePermissions(permissions, walkparams, accdesc);

bit r, w;
bit or, ow;

if accdesc.acctype == AccessType\_TTW then
    r = s2perms.r_mmu;
    w = s2perms.w_mmu;
    or = s2perms.or_mmu;
    ow = s2perms.ow_mmu;
elseif accdesc.rcw then
    r = s2perms.r_rcw;
    w = s2perms.w_rcw;
    or = s2perms.or_rcw;
    ow = s2perms.ow_rcw;
else
    r = s2perms.r;
    w = s2perms.w;
    or = s2perms.or;
    ow = s2perms.ow;

if accdesc.acctype == AccessType\_TTW then
    if (accdesc.toplevel && accdesc.varange == VRange\_LOWER &&
        ((walkparams.tl0 == '1' && s2perms.toplevel0 == '0') ||
         (walkparams.tl1 == '1' && s2perms.<toplevel1,toplevel0> == '10'))) then
        fault.statuscode = Fault\_Permission;
        fault.toplevel = TRUE;
    elseif (accdesc.toplevel && accdesc.varange == VRange\_UPPER &&
        ((walkparams.tl1 == '1' && s2perms.toplevel1 == '0') ||
         (walkparams.tl0 == '1' && s2perms.<toplevel1,toplevel0> == '01'))) then
        fault.statuscode = Fault\_Permission;
        fault.toplevel = TRUE;
    // Stage 2 Permission fault due to AssuredOnly check
    elseif (walkstate.s2assuredonly == '1' && !ipa.slassured) then
        fault.statuscode = Fault\_Permission;
        fault.assuredonly = TRUE;

    elseif walkparams.ptw == '1' && memtype == MemType\_Device then
        fault.statuscode = Fault\_Permission;
    elseif s2perms.overlay && or == '0' then
        fault.statuscode = Fault\_Permission;
        fault.overlay = TRUE;
    elseif accdesc.write && s2perms.overlay && ow == '0' then
        fault.statuscode = Fault\_Permission;
        fault.overlay = TRUE;
    // Prevent translation table walks in Non-secure space by Realm state
    elseif accdesc.ss == SS\_Realm && walkstate.baseaddress.paspace != PAS\_Realm then
        fault.statuscode = Fault\_Permission;
    elseif r == '0' then
        fault.statuscode = Fault\_Permission;
    elseif accdesc.write && w == '0' then
        fault.statuscode = Fault\_Permission;
    elseif (accdesc.write && !(walkparams.<ha,hd> == '11') && walkparams.s2pie == '1' &&
        permissions.s2dirty == '0') then
        fault.statuscode = Fault\_Permission;
        fault.dirtybit = TRUE;

    // Stage 2 Permission fault due to AssuredOnly check
    elseif ((walkstate.s2assuredonly == '1' && !ipa.slassured) ||
        (walkstate.s2assuredonly != '1' && HaveGCS() && VTCR_EL2.GCSH == '1' &&

```

```

        accdesc.acctype == AccessType\_GCS && accdesc.el != EL0)) then
    fault.statuscode = Fault\_Permission;
    fault.assuredonly = TRUE;

elseif accdesc.acctype == AccessType\_IFETCH then
    if s2perms.overlay && s2perms.ox == '0' then
        fault.statuscode = Fault\_Permission;
        fault.overlay = TRUE;
    elseif (memtype == MemType\_Device &&
        ConstrainUnpredictable(Unpredictable\_INSTRDEVICE) == Constraint\_FAULT) then
        fault.statuscode = Fault\_Permission;

    // Prevent execution from Non-secure space by Realm state
    elseif accdesc.ss == SS\_Realm && walkstate.baseaddress.paspace != PAS\_Realm then
        fault.statuscode = Fault\_Permission;
    elseif s2perms.x == '0' then
        fault.statuscode = Fault\_Permission;

elseif accdesc.acctype == AccessType\_DC then
    if accdesc.cacheop == CacheOp\_Invalidate then
        if !ELUsingAArch32(EL1) && s2perms.overlay && ow == '0' then
            fault.statuscode = Fault\_Permission;
            fault.overlay = TRUE;
        if !ELUsingAArch32(EL1) && w == '0' then
            fault.statuscode = Fault\_Permission;
    elseif !ELUsingAArch32(EL1) && accdesc.el == EL0 && s2perms.overlay && or == '0' then
        fault.statuscode = Fault\_Permission;
        fault.overlay = TRUE;
    elseif (walkparams.cmow == '1' &&
        accdesc.opscope == CacheOpScope\_PoC &&
        accdesc.cacheop == CacheOp\_CleanInvalidate &&
        s2perms.overlay && ow == '0') then
        fault.statuscode = Fault\_Permission;
        fault.overlay = TRUE;
    elseif !ELUsingAArch32(EL1) && accdesc.el == EL0 && r == '0' then
        fault.statuscode = Fault\_Permission;
    elseif (walkparams.cmow == '1' &&
        accdesc.opscope == CacheOpScope\_PoC &&
        accdesc.cacheop == CacheOp\_CleanInvalidate &&
        w == '0') then
        fault.statuscode = Fault\_Permission;

elseif accdesc.acctype == AccessType\_IC then
    if (!ELUsingAArch32(EL1) && accdesc.el == EL0 && s2perms.overlay && or == '0' &&
        boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
        fault.statuscode = Fault\_Permission;
        fault.overlay = TRUE;
    elseif walkparams.cmow == '1' && s2perms.overlay && ow == '0' then
        fault.statuscode = Fault\_Permission;
        fault.overlay = TRUE;
    elseif (!ELUsingAArch32(EL1) && accdesc.el == EL0 && r == '0' &&
        boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
        fault.statuscode = Fault\_Permission;
    elseif walkparams.cmow == '1' && w == '0' then
        fault.statuscode = Fault\_Permission;

elseif accdesc.read && s2perms.overlay && or == '0' then
    fault.statuscode = Fault\_Permission;
    fault.overlay = TRUE;
    fault.write = FALSE;
elseif accdesc.write && s2perms.overlay && ow == '0' then
    fault.statuscode = Fault\_Permission;
    fault.overlay = TRUE;
    fault.write = TRUE;
elseif accdesc.read && r == '0' then
    fault.statuscode = Fault\_Permission;
    fault.write = FALSE;
elseif accdesc.write && w == '0' then
    fault.statuscode = Fault\_Permission;
    fault.write = TRUE;

```

```

elseif ((accdesc.tagaccess || accdesc.tagchecked) &&
        ipa.memattrs.tags == MemTag\_AllocationTagged &&
        permissions.s2tag_na == '1' && S2DCacheEnabled()) then
    fault.statuscode = Fault\_Permission;
    fault.tagaccess = TRUE;
    fault.write = accdesc.tagaccess && accdesc.write;
elseif (accdesc.write && !(walkparams.<ha,hd> == '11') && walkparams.s2pie == '1' &&
        permissions.s2dirty == '0') then
    fault.statuscode = Fault\_Permission;
    fault.dirtybit = TRUE;
    fault.write = TRUE;

// MRO* allows only RCW and MMU writes
boolean mro;
if s2perms.overlay then
    mro = (s2perms.<w,w_rcw,w_mmu> AND s2perms.<ow,ow_rcw,ow_mmu>) == '011';
else
    mro = s2perms.<w,w_rcw,w_mmu> == '011';

return (fault, mro);

```

### Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S2ComputePermissions

```

// AArch64.S2ComputePermissions()
// =====
// Compute the overall stage 2 permissions.

S2AccessControls AArch64.S2ComputePermissions(Permissions permissions, S2TTWParams walkparams,
        AccessDescriptor accdesc)

S2AccessControls s2perms;

if walkparams.s2pie == '1' then
    s2perms = AArch64.S2IndirectBasePermissions(permissions, accdesc);
    s2perms.overlay = HaveS2POExt() && VTCR_EL2.S2POE == '1';
    if s2perms.overlay then
        s2overlay_perms = AArch64.S2OverlayPermissions(permissions, accdesc);
        s2perms.or = s2overlay_perms.or;
        s2perms.ow = s2overlay_perms.ow;
        s2perms.ox = s2overlay_perms.ox;
        s2perms.or_rcw = s2overlay_perms.or_rcw;
        s2perms.ow_rcw = s2overlay_perms.ow_rcw;
        s2perms.or_mmu = s2overlay_perms.or_mmu;
        s2perms.ow_mmu = s2overlay_perms.ow_mmu;

        // Toplevel is applicable only when the effective S2 permissions is MRO
        if ((s2perms.<w,w_rcw,w_mmu> AND s2perms.<ow,ow_rcw,ow_mmu>) == '011') then
            s2perms.toplevel0 = s2perms.toplevel0 OR s2overlay_perms.toplevel0;
            s2perms.toplevel1 = s2perms.toplevel1 OR s2overlay_perms.toplevel1;

        else
            s2perms.toplevel0 = '0';
            s2perms.toplevel1 = '0';
    else
        s2perms = AArch64.S2DirectBasePermissions(permissions, accdesc);

return s2perms;

```



## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S2DirectBasePermissions

```
// AArch64.S2DirectBasePermissions()
// =====
// Computes the stage 2 direct base permissions.

S2AccessControls AArch64.S2DirectBasePermissions(Permissions permissions,
AccessDescriptor accdesc)

    S2AccessControls s2perms;
    r = permissions.s2ap<0>;
    w = permissions.s2ap<1>;
    bit px, ux;
    case (permissions.s2xn:permissions.s2xnx) of
        when '00' (px,ux) = ('1','1');
        when '01' (px,ux) = ('0','1');
        when '10' (px,ux) = ('0','0');
        when '11' (px,ux) = ('1','0');

    x = if accdesc.el == EL0 then ux else px;
    s2perms.r = r;
    s2perms.w = w;
    s2perms.x = x;
    s2perms.r_rcw = r;
    s2perms.w_rcw = w;
    s2perms.r_mmu = r;
    s2perms.w_mmu = w;

    return s2perms;
```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S2HasAlignmentFault

```
// AArch64.S2HasAlignmentFault()
// =====
// Returns whether stage 2 output fails alignment requirement on data accesses
// to Device memory

boolean AArch64.S2HasAlignmentFault(AccessDescriptor accdesc, boolean aligned,
MemoryAttributes memattrs)

    if accdesc.acctype == AccessType\_IFETCH then
        return FALSE;
    elsif HaveMTEExt() && accdesc.tagaccess && accdesc.write then
        return (memattrs.memtype == MemType\_Device &&
            ConstrainUnpredictable(Unpredictable\_DEVICETAGSTORE) == Constraint\_FAULT);
    elsif accdesc.acctype == AccessType\_DCZero then
        return memattrs.memtype == MemType\_Device;
    else
        return memattrs.memtype == MemType\_Device && !aligned;
```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S2InconsistentSL

```
// AArch64.S2InconsistentSL()
// =====
// Detect inconsistent configuration of stage 2 TxSZ and SL fields

boolean AArch64.S2InconsistentSL(S2TTWParams walkparams)
    startlevel    = AArch64.S2StartLevel(walkparams);
    levels        = FINAL_LEVEL - startlevel;
    granulebits   = TGxGranuleBits(walkparams.tgx);
    descsize_log2 = 3;
    stride        = granulebits - descsize_log2;

    // Input address size must at least be large enough to be resolved from the start level
    sl_min_iasize = (
        levels * stride // Bits resolved by table walk, except initial level
        + granulebits   // Bits directly mapped to output address
        + 1);           // At least 1 more bit to be decoded by initial level

    // Can accomodate 1 more stride in the level + concatenation of up to 2^4 tables
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    iasize        = AArch64.IASize(walkparams.txsz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S2IndirectBasePermissions

```
// AArch64.S2IndirectBasePermissions()
// =====
// Computes the stage 2 indirect base permissions.

S2AccessControls AArch64.S2IndirectBasePermissions(Permissions permissions,
AccessDescriptor accdesc)

    bit r, w;
    bit r_rcw, w_rcw;
    bit r_mmu, w_mmu;
    bit px, ux;
    bit toplevel0, toplevel1;
    S2AccessControls s2perms;

    bits(4) s2pi = permissions.s2pi;
    case s2pi of
        when '0000' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0'); // No Access
        when '0001' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0'); // Reserved
        when '0010' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1'); // MRO
        when '0011' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1'); // MRO-TL1
        when '0100' (r,w,px,ux,w_rcw,w_mmu) = ('0','1','0','0','0','0'); // Write Only
        when '0101' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0'); // Reserved
        when '0110' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1'); // MRO-TL0
        when '0111' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1'); // MRO-TL01
        when '1000' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','0'); // Read Only
        when '1001' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','1','0','0'); // Read, Unpriv Execute
        when '1010' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','0','0','0'); // Read, Priv Execute
        when '1011' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','1','0','0'); // Read, All Execute
        when '1100' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','0','1','1'); // RW
        when '1101' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','1','1','1'); // RW, Unpriv Execute
        when '1110' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','0','1','1'); // RW, Priv Execute
        when '1111' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','1','1','1'); // RW, All Execute

    x = if accdesc.el == EL0 then ux else px;

    // RCW and MMU read permissions.
    (r_rcw, r_mmu) = (r, r);

    // Stage 2 Top Level Permission Attributes.
    case s2pi of
        when '0110' (toplevel0,toplevel1) = ('1','0');
        when '0011' (toplevel0,toplevel1) = ('0','1');
        when '0111' (toplevel0,toplevel1) = ('1','1');
        otherwise (toplevel0,toplevel1) = ('0','0');

    s2perms.r = r;
    s2perms.w = w;
    s2perms.x = x;
    s2perms.r_rcw = r_rcw;
    s2perms.r_mmu = r_mmu;
    s2perms.w_rcw = w_rcw;
    s2perms.w_mmu = w_mmu;
    s2perms.toplevel0 = toplevel0;
    s2perms.toplevel1 = toplevel1;

    return s2perms;
```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S2InvalidSL

```
// AArch64.S2InvalidSL()
// =====
// Detect invalid configuration of SL field

boolean AArch64.S2InvalidSL(S2TTWParams walkparams)
  case walkparams.tgx of
    when TGx\_4KB
      case walkparams.sl2:walkparams.sl0 of
        when '1x1' return TRUE;
        when '11x' return TRUE;
        when '010' return AArch64.PAMax() < 44;
        when '011' return !HaveSmallTranslationTableExt();
        otherwise return FALSE;
    when TGx\_16KB
      case walkparams.sl0 of
        when '11' return walkparams.ds == '0';
        when '10' return AArch64.PAMax() < 42;
        otherwise return FALSE;
    when TGx\_64KB
      case walkparams.sl0 of
        when '11' return TRUE;
        when '10' return AArch64.PAMax() < 44;
        otherwise return FALSE;
```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S2OverlayPermissions

```
// AArch64.S2OverlayPermissions()
// =====
// Computes the stage 2 overlay permissions.

S2AccessControls AArch64.S2OverlayPermissions(Permissions permissions, AccessDescriptor accdesc)
    bit r, w;
    bit r_rcw, w_rcw;
    bit r_mmu, w_mmu;
    bit px, ux;
    bit toplevel0, toplevel1;
    S2AccessControls s2overlay_perms;

    integer index = 4 * UInt(permissions.s2po_index);
    bits(4) s2po = S2POR_EL1[index+3 : index];
    case s2po of
        when '0000' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0'); // No Access
        when '0001' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0'); // Reserved
        when '0010' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1'); // MRO
        when '0011' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1'); // MRO-TL1
        when '0100' (r,w,px,ux,w_rcw,w_mmu) = ('0','1','0','0','0','0'); // Write Only
        when '0101' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0'); // Reserved
        when '0110' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1'); // MRO-TL0
        when '0111' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1'); // MRO-TL01
        when '1000' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','0'); // Read Only
        when '1001' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','1','0','0'); // Read, Unpriv Execute
        when '1010' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','0','0','0'); // Read, Priv Execute
        when '1011' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','1','0','0'); // Read, All Execute
        when '1100' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','0','1','1'); // RW
        when '1101' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','1','1','1'); // RW, Unpriv Execute
        when '1110' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','0','1','1'); // RW, Priv Execute
        when '1111' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','1','1','1'); // RW, All Execute

    x = if accdesc.el == EL0 then ux else px;

    // RCW and MMU read permissions.
    (r_rcw, r_mmu) = (r, r);

    // Stage 2 Top Level Permission Attributes.
    case s2po of
        when '0110' (toplevel0,toplevel1) = ('1','0');
        when '0011' (toplevel0,toplevel1) = ('0','1');
        when '0111' (toplevel0,toplevel1) = ('1','1');
        otherwise (toplevel0,toplevel1) = ('0','0');

    s2overlay_perms.or = r;
    s2overlay_perms.ow = w;
    s2overlay_perms.ox = x;
    s2overlay_perms.or_rcw = r_rcw;
    s2overlay_perms.ow_rcw = w_rcw;
    s2overlay_perms.or_mmu = r_mmu;
    s2overlay_perms.ow_mmu = w_mmu;
    s2overlay_perms.toplevel0 = toplevel0;
    s2overlay_perms.toplevel1 = toplevel1;

    return s2overlay_perms;
```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S2TxSZFaults

```
// AArch64.S2TxSZFaults()
// =====
// Detect whether configuration of stage 2 TxSZ field generates a fault

boolean AArch64.S2TxSZFaults(S2TTWParams walkparams, boolean slaarch64)
    mintxs = AArch64.S2MinTxSZ(walkparams.d128, walkparams.ds, walkparams.tgx, slaarch64);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);

    if UInt(walkparams.txsz) < mintxs then
        return (Have52BitPAExt() ||
            boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum");
    if UInt(walkparams.txsz) > maxtxsz then
        return boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum";

    return FALSE;
```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.VAIsOutOfRange

```
// AArch64.VAIsOutOfRange()
// =====
// Check bits not resolved by translation are identical and of accepted value

boolean AArch64.VAIsOutOfRange(bits(64) va_in, AccessType acctype,
                                Regime regime, S1TTWParams walkparams)
    bits(64) va = va_in;

    addrtop = AArch64.AddrTop(walkparams.tbid, acctype, walkparams.tbi);

    // If the VA has a Logical Address Tag then the bits holding the Logical Address Tag are
    // ignored when checking if the address is out of range.
    if walkparams.mtx == '1' then
        va<59:56> = if AArch64.GetVARange(va) == VARange\_UPPER then '1111' else '0000';

    // Input Address size
    iasize = AArch64.IASize(walkparams.txsz);

    // The min value of TxSZ can be 8, with LVA3 implemented.
    // If TxSZ is set to 8 iasize becomes 64 - 8 = 56
    // If tbi is also set, addrtop becomes 55
    // Then the return statements check va<56:55>
    // The check here is to guard against this corner case.
    if addrtop < iasize then
        return FALSE;

    if HasUnprivileged(regime) then
        if AArch64.GetVARange(va) == VARange\_LOWER then
            return IsZero(va<addrtop:iasize>);
        else
            return IsOnes(va<addrtop:iasize>);
    else
        return IsZero(va<addrtop:iasize>);
```



```

// AArch64.S2ApplyFWBMemAttrs()
// =====
// Apply stage 2 forced Write-Back on stage 1 memory attributes.

MemoryAttributes AArch64.S2ApplyFWBMemAttrs(MemoryAttributes s1_memattrs, S2TTWParams walkparams,
                                             bits(N) descriptor)

    MemoryAttributes memattrs;
    s2_attr = descriptor<5:2>;
    s2_sh   = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
    s2_fnxs = descriptor<11>;

    if s2_attr<2> == '0' then          // S2 Device, S1 any
        s2_device = DecodeDevice(s2_attr<1:0>);
        memattrs.memtype = MemType Device;
        if s1_memattrs.memtype == MemType Device then
            memattrs.device = S2CombineS1Device(s1_memattrs.device, s2_device);
        else
            memattrs.device = s2_device;

        memattrs.xs = s1_memattrs.xs;

    elseif s2_attr<1:0> == '11' then    // S2 attr = S1 attr
        memattrs = s1_memattrs;

    elseif s2_attr<1:0> == '10' then    // Force writeback
        memattrs.memtype = MemType Normal;
        memattrs.inner.attrs = MemAttr WB;
        memattrs.outer.attrs = MemAttr WB;

        if (s1_memattrs.memtype == MemType Normal &&
            s1_memattrs.inner.attrs != MemAttr NC) then
            memattrs.inner.hints      = s1_memattrs.inner.hints;
            memattrs.inner.transient = s1_memattrs.inner.transient;
        else
            memattrs.inner.hints      = MemHint RWA;
            memattrs.inner.transient = FALSE;

        if (s1_memattrs.memtype == MemType Normal &&
            s1_memattrs.outer.attrs != MemAttr NC) then
            memattrs.outer.hints      = s1_memattrs.outer.hints;
            memattrs.outer.transient = s1_memattrs.outer.transient;
        else
            memattrs.outer.hints      = MemHint RWA;
            memattrs.outer.transient = FALSE;

        memattrs.xs = '0';

    else                                // Non-cacheable unless S1 is device
        if s1_memattrs.memtype == MemType Device then
            memattrs = s1_memattrs;
        else
            MemAttrHints cacheability_attr;
            cacheability_attr.attrs = MemAttr NC;

            memattrs.memtype = MemType Normal;
            memattrs.inner   = cacheability_attr;
            memattrs.outer   = cacheability_attr;

            memattrs.xs = s1_memattrs.xs;

    s2_shareability = DecodeShareability(s2_sh);
    memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability, s2_shareability);
    memattrs.tags        = S2MemTagType(memattrs, s1_memattrs.tags);
    memattrs.notagaccess = (s2_attr<3:1> == '111' && memattrs.tags == MemTag AllocationTagged);

    if s2_fnxs == '1' then
        memattrs.xs = '0';

    memattrs.shareability = EffectiveShareability(memattrs);
    return memattrs;

```



## Library pseudocode for aarch64/translation/vmsa\_tlbcontext/AArch64.GetS1TLBContext

```
// AArch64.GetS1TLBContext()
// =====
// Gather translation context for accesses with VA to match against TLB entries

TLBContext AArch64.GetS1TLBContext(Regime regime, SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    case regime of
        when Regime\_EL3    tlbcontext = AArch64.TLBContextEL3(ss, va, tg);
        when Regime\_EL2    tlbcontext = AArch64.TLBContextEL2(ss, va, tg);
        when Regime\_EL20   tlbcontext = AArch64.TLBContextEL20(ss, va, tg);
        when Regime\_EL10   tlbcontext = AArch64.TLBContextEL10(ss, va, tg);

    tlbcontext.includes_s1 = TRUE;
    // The following may be amended for EL1&0 Regime if caching of stage 2 is successful
    tlbcontext.includes_s2 = FALSE;
    // The following may be amended if Granule Protection Check passes
    tlbcontext.includes_gpt = FALSE;
    return tlbcontext;
```

## Library pseudocode for aarch64/translation/vmsa\_tlbcontext/AArch64.GetS2TLBContext

```
// AArch64.GetS2TLBContext()
// =====
// Gather translation context for accesses with IPA to match against TLB entries

TLBContext AArch64.GetS2TLBContext(SecurityState ss, FullAddress ipa, TGx tg)
    assert EL2Enabled();

    TLBContext tlbcontext;

    tlbcontext.ss          = ss;
    tlbcontext.regime      = Regime\_EL10;
    tlbcontext.ipaspace    = ipa.paspace;
    tlbcontext.vmid        = VMID[];
    tlbcontext.tg          = tg;
    tlbcontext.ia          = ZeroExtend(ipa.address, 64);
    if HaveCommonNotPrivateTransExt() then
        tlbcontext.cnp = if ipa.paspace == PAS\_Secure then VSTTBR_EL2.CnP else VTTBR_EL2.CnP;
    else
        tlbcontext.cnp = '0';

    tlbcontext.includes_s1 = FALSE;
    tlbcontext.includes_s2 = TRUE;
    // This may be amended if Granule Protection Check passes
    tlbcontext.includes_gpt = FALSE;
    return tlbcontext;
```

## Library pseudocode for aarch64/translation/vmsa\_tlbcontext/AArch64.TLBContextEL10

```
// AArch64.TLBContextEL10()
// =====
// Gather translation context for accesses under EL10 regime to match against TLB entries

TLBContext AArch64.TLBContextEL10(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss      = ss;
    tlbcontext.regime  = Regime\_EL10;
    tlbcontext.vmid    = VMID[];
    tlbcontext.asid    = if TCR_EL1.A1 == '0' then TTBR0_EL1.ASID else TTBR1_EL1.ASID;
    if TCR_EL1.AS == '0' then
        tlbcontext.asid<15:8> = Zeros(8);
    tlbcontext.tg      = tg;
    tlbcontext.ia      = va;

    if HaveCommonNotPrivateTransExt() then
        if AArch64.GetVARange(va) == VARange\_LOWER then
            tlbcontext.cnp = TTBR0_EL1.CnP;
        else
            tlbcontext.cnp = TTBR1_EL1.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

## Library pseudocode for aarch64/translation/vmsa\_tlbcontext/AArch64.TLBContextEL2

```
// AArch64.TLBContextEL2()
// =====
// Gather translation context for accesses under EL2 regime to match against TLB entries

TLBContext AArch64.TLBContextEL2(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss      = ss;
    tlbcontext.regime  = Regime\_EL2;
    tlbcontext.tg      = tg;
    tlbcontext.ia      = va;
    tlbcontext.cnp      = if HaveCommonNotPrivateTransExt() then TTBR0_EL2.CnP else '0';

    return tlbcontext;
```

## Library pseudocode for aarch64/translation/vmsa\_tlbcontext/AArch64.TLBContextEL20

```
// AArch64.TLBContextEL20()
// =====
// Gather translation context for accesses under EL20 regime to match against TLB entries

TLBContext AArch64.TLBContextEL20(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss      = ss;
    tlbcontext.regime  = Regime\_EL20;
    tlbcontext.asid    = if TCR_EL2.A1 == '0' then TTBR0_EL2.ASID else TTBR1_EL2.ASID;
    if TCR_EL2.AS == '0' then
        tlbcontext.asid<15:8> = Zeros(8);
    tlbcontext.tg      = tg;
    tlbcontext.ia      = va;

    if HaveCommonNotPrivateTransExt() then
        if AArch64.GetVARange(va) == VARange\_LOWER then
            tlbcontext.cnp = TTBR0_EL2.CnP;
        else
            tlbcontext.cnp = TTBR1_EL2.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

## Library pseudocode for aarch64/translation/vmsa\_tlbcontext/AArch64.TLBContextEL3

```
// AArch64.TLBContextEL3()
// =====
// Gather translation context for accesses under EL3 regime to match against TLB entries

TLBContext AArch64.TLBContextEL3(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss      = ss;
    tlbcontext.regime  = Regime\_EL3;
    tlbcontext.tg      = tg;
    tlbcontext.ia      = va;
    tlbcontext.cnp     = if HaveCommonNotPrivateTransExt() then TTBR0_EL3.CnP else '0';

    return tlbcontext;
```

## Library pseudocode for aarch64/translation/vmsa\_translation/AArch64.FullTranslate

```
// AArch64.FullTranslate()
// =====
// Address translation as specified by VMSA
// Alignment check NOT due to memory type is expected to be done before translation

AddressDescriptor AArch64.FullTranslate(bits(64) va, AccessDescriptor accdesc, boolean aligned)
    Regime regime = TranslationRegime(accdesc.el);
    FaultRecord fault = NoFault(accdesc);

    AddressDescriptor ipa;
    (fault, ipa) = AArch64.S1Translate(fault, regime, va, aligned, accdesc);

    if fault.statuscode != Fault\_None then
        return CreateFaultyAddressDescriptor(va, fault);

    if accdesc.ss == SS\_Realm then
        assert EL2Enabled();
    if regime == Regime\_EL10 && EL2Enabled() then
        slaarch64 = TRUE;
        AddressDescriptor pa;
        (fault, pa) = AArch64.S2Translate(fault, ipa, slaarch64, aligned, accdesc);

        if fault.statuscode != Fault\_None then
            return CreateFaultyAddressDescriptor(va, fault);
        else
            return pa;
    else
        return ipa;
```

## Library pseudocode for aarch64/translation/vmsa\_translation/AArch64.MemSwapTableDesc

```
// AArch64.MemSwapTableDesc()
// =====
// Perform HW update of table descriptor as an atomic operation

(FaultRecord, bits(N)) AArch64.MemSwapTableDesc(FaultRecord fault_in, bits(N) prev_desc,
                                                bits(N) new_desc, bit ee,
                                                AccessDescriptor descaccess,
                                                AddressDescriptor descpaddr)

FaultRecord fault = fault_in;
boolean iswrite;

if HaveRME() then
    fault.gpcf = GranuleProtectionCheck(descpaddr, descaccess);
    if fault.gpcf.gpf != GPCF None then
        fault.statuscode = Fault GPCFOnWalk;
        fault.paddress = descpaddr.paddress;
        fault.gpcfs2walk = fault.secondstage;
        return (fault, bits(N) UNKNOWN);

// All observers in the shareability domain observe the
// following memory read and write accesses atomically.
bits(N) mem_desc;
PhysMemRetStatus memstatus;
(memstatus, mem_desc) = PhysMemRead(descpaddr, N DIV 8, descaccess);

if ee == '1' then
    mem_desc = BigEndianReverse(mem_desc);

if IsFault(memstatus) then
    iswrite = FALSE;
    fault = HandleExternalTTWAbort(memstatus, iswrite, descpaddr, descaccess, N DIV 8, fault);
    if IsFault(fault.statuscode) then
        return (fault, bits(N) UNKNOWN);

if mem_desc == prev_desc then
    ordered_new_desc = if ee == '1' then BigEndianReverse(new_desc) else new_desc;
    memstatus = PhysMemWrite(descpaddr, N DIV 8, descaccess, ordered_new_desc);

    if IsFault(memstatus) then
        iswrite = TRUE;
        fault = HandleExternalTTWAbort(memstatus, iswrite, descpaddr, descaccess, N DIV 8,
                                         fault);

        if IsFault(fault.statuscode) then
            return (fault, bits(N) UNKNOWN);

// Reflect what is now in memory (in little endian format)
mem_desc = new_desc;

return (fault, mem_desc);
```



```

// AArch64.S1DisabledOutput()
// =====
// Map the VA to IPA/PA and assign default memory attributes

(FaultRecord, AddressDescriptor) AArch64.S1DisabledOutput(FaultRecord fault_in, Regime regime,
                                                         bits(64) va_in, AccessDescriptor accdesc,
                                                         boolean aligned)

bits(64) va = va_in;
walkparams = AArch64.GetS1TTWParams(regime, accdesc.ss, va);
FaultRecord fault = fault_in;

// No memory page is guarded when stage 1 address translation is disabled
SetInGuardedPage(FALSE);

// Output Address
FullAddress oa;
oa.address = va<55:0>;
case accdesc.ss of
    when SS\_Secure      oa.paspace = PAS\_Secure;
    when SS\_NonSecure   oa.paspace = PAS\_NonSecure;
    when SS\_Root        oa.paspace = PAS\_Root;
    when SS\_Realm       oa.paspace = PAS\_Realm;

MemoryAttributes memattrs;
if regime == Regime\_EL10 && EL2Enabled() && walkparams.dc == '1' then
    MemAttrHints default_cacheability;
    default_cacheability.attrs = MemAttr\_WB;
    default_cacheability.hints = MemHint\_RWA;
    default_cacheability.transient = FALSE;

    memattrs.memtype = MemType\_Normal;
    memattrs.outer = default_cacheability;
    memattrs.inner = default_cacheability;
    memattrs.shareability = Shareability\_NSH;
    if walkparams.dct == '1' then
        memattrs.tags = MemTag\_AllocationTagged;
    elsif walkparams.mtx == '1' then
        memattrs.tags = MemTag\_CanonicalyTagged;
    else
        memattrs.tags = MemTag\_Untagged;
    memattrs.xs = '0';
elsif accdesc.acctype == AccessType\_IFETCH then
    MemAttrHints i_cache_attr;
    if AArch64.S1ICacheEnabled(regime) then
        i_cache_attr.attrs = MemAttr\_WT;
        i_cache_attr.hints = MemHint\_RA;
        i_cache_attr.transient = FALSE;
    else
        i_cache_attr.attrs = MemAttr\_NC;

    memattrs.memtype = MemType\_Normal;
    memattrs.outer = i_cache_attr;
    memattrs.inner = i_cache_attr;
    memattrs.shareability = Shareability\_OSH;
    memattrs.tags = MemTag\_Untagged;
    memattrs.xs = '1';
else
    memattrs.memtype = MemType\_Device;
    memattrs.device = DeviceType\_nGnRnE;
    memattrs.shareability = Shareability\_OSH;
    if walkparams.mtx == '1' then
        memattrs.tags = MemTag\_CanonicalyTagged;
    else
        memattrs.tags = MemTag\_Untagged;
    memattrs.xs = '1';
memattrs.notagaccess = FALSE;

if walkparams.mtx == '1' && walkparams.tbi == '0' && accdesc.acctype != AccessType\_IFETCH then
    // For the purpose of the checks in this function, the MTE tag bits are ignored.

```

```

    va<59:56> = if HasUnprivileged(regime) then Replicate(va<55>, 4) else '0000';

    fault.level = 0;
    addrtop      = AArch64.AddrTop(walkparams.tbid, accdesc.acctype, walkparams.tbi);

    if !IsZero(va<addrtop:AArch64.PAMax()>) then
        fault.statuscode = Fault\_AddressSize;
    elsif AArch64.S1HasAlignmentFault(accdesc, aligned, walkparams.ntlsmid, memattrs) then
        fault.statuscode = Fault\_Alignment;

    if fault.statuscode != Fault\_None then
        return (fault, AddressDescriptor UNKNOWN);
    else
        ipa = CreateAddressDescriptor(va_in, oa, memattrs);
        ipa.mecid = AArch64.S1DisabledOutputMECID(walkparams, regime, ipa.paddress.paspace);
        return (fault, ipa);

```





```

// AArch64.S1Translate()
// =====
// Translate VA to IPA/PA depending on the regime

(FaultRecord, AddressDescriptor) AArch64.S1Translate(FaultRecord fault_in, Regime regime,
                                                    bits(64) va, boolean aligned,
                                                    AccessDescriptor accdesc)

FaultRecord fault = fault_in;
// Prepare fault fields in case a fault is detected
fault.secondstage = FALSE;
fault.s2fslwalk   = FALSE;

if !AArch64.S1Enabled(regime, accdesc.acctype) then
    return AArch64.S1DisabledOutput(fault, regime, va, accdesc, aligned);

walkparams = AArch64.GetS1TTWParams(regime, accdesc.ss, va);

constant integer slmintxsz = AArch64.S1MinTxSZ(regime, walkparams.dl28,
                                             walkparams.ds, walkparams.tgx);
constant integer slmaxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
if AArch64.S1TxSZFaults(regime, walkparams) then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AccessDescriptor UNKNOWN);
elseif UInt(walkparams.txsz) < slmintxsz then
    walkparams.txsz = slmintxsz<5:0>;
elseif UInt(walkparams.txsz) > slmaxtxsz then
    walkparams.txsz = slmaxtxsz<5:0>;

if AArch64.VAIsOutOfRange(va, accdesc.acctype, regime, walkparams) then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AccessDescriptor UNKNOWN);

if accdesc.el == ELO && walkparams.e0pd == '1' then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AccessDescriptor UNKNOWN);

if HaveTME() && accdesc.el == ELO && walkparams.nfd == '1' && accdesc.transactional then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AccessDescriptor UNKNOWN);

if HaveSVE() && accdesc.el == ELO && walkparams.nfd == '1' && (
    (accdesc.nonfault && accdesc.contiguous) ||
    (accdesc.firstfault && !accdesc.first && !accdesc.contiguous)) then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AccessDescriptor UNKNOWN);

AccessDescriptor descipaddr;
TTWState walkstate;
bits(128) descriptor;
bits(128) new_desc;
bits(128) mem_desc;
repeat
    if walkparams.dl28 == '1' then
        (fault, descipaddr, walkstate, descriptor) = AArch64.S1Walk(fault, walkparams, va,
                                                                    regime, accdesc, 128);
    else
        (fault, descipaddr, walkstate, descriptor<63:0>) = AArch64.S1Walk(fault, walkparams,
                                                                    va, regime, accdesc,
                                                                    64);

        descriptor<127:64> = Zeros(64);
    if fault.statuscode != Fault\_None then
        return (fault, AccessDescriptor UNKNOWN);

if accdesc.acctype == AccessType\_IFETCH then
    // Flag the fetched instruction is from a guarded page

```

```

    SetInGuardedPage(walkstate.guardedpage == '1');

    if AArch64.S1HasAlignmentFault(accdesc, aligned, walkparams.ntlsmid,
        walkstate.memattrs) then
        fault.statuscode = Fault\_Alignment;

    if fault.statuscode == Fault\_None then
        fault = AArch64.S1CheckPermissions(fault, regime, walkstate, walkparams, accdesc);

    new_desc = descriptor;
    if walkparams.ha == '1' && AArch64.SettingAccessFlagPermitted(fault) then
        // Set descriptor AF bit
        new_desc<10> = '1';

    // If HW update of dirty bit is enabled, the walk state permissions
    // will already reflect a configuration permitting writes.
    // The update of the descriptor occurs only if the descriptor bits in
    // memory do not reflect that and the access instigates a write.

    if (AArch64.SettingDirtyStatePermitted(fault) &&
        walkparams.ha == '1' &&
        walkparams.hd == '1' &&
        (walkparams.pie == '1' || descriptor<51> == '1') &&
        accdesc.write &&
        !(accdesc.acctype IN {AccessType\_AT, AccessType\_IC, AccessType\_DC})) then
        // Clear descriptor AP[2]/nDirty bit permitting stage 1 writes
        new_desc<7> = '0';

    // Either the access flag was clear or AP[2]/nDirty is set
    if new_desc != descriptor then
        AddressDescriptor descpaddr;
        descaccess = CreateAccDescTTEUpdate(accdesc);
        if regime == Regime\_EL10 && EL2Enabled() then
            FaultRecord s2fault;
            slaarch64 = TRUE;
            s2aligned = TRUE;
            (s2fault, descpaddr) = AArch64.S2Translate(fault, descipaddr, slaarch64, s2aligned,
                descaccess);

            if s2fault.statuscode != Fault\_None then
                return (s2fault, AddressDescriptor UNKNOWN);

        else
            descpaddr = descipaddr;
            if walkparams.d128 == '1' then
                (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
                    walkparams.ee, descaccess, descpaddr);
            else
                (fault, mem_desc<63:0>) = AArch64.MemSwapTableDesc(fault, descriptor<63:0>,
                    new_desc<63:0>, walkparams.ee,
                    descaccess, descpaddr);

                mem_desc<127:64> = Zeros(64);

    until new_desc == descriptor || mem_desc == new_desc;

    if fault.statuscode != Fault\_None then
        return (fault, AddressDescriptor UNKNOWN);

    // Output Address
    oa = StageOA(va, walkparams.d128, walkparams.tgxx, walkstate);
    MemoryAttributes memattrs;
    if (accdesc.acctype == AccessType\_IFETCH &&
        (walkstate.memattrs.memtype == MemType\_Device || !AArch64.S1ICacheEnabled(regime))) then
        // Treat memory attributes as Normal Non-Cacheable
        memattrs = NormalNCMemAttr();
        memattrs.xs = walkstate.memattrs.xs;
    elsif (accdesc.acctype != AccessType\_IFETCH && !AArch64.S1DCacheEnabled(regime) &&
        walkstate.memattrs.memtype == MemType\_Normal) then
        // Treat memory attributes as Normal Non-Cacheable
        memattrs = NormalNCMemAttr();

```

```

memattrs.xs = walkstate.memattrs.xs;

// The effect of SCTLR_ELx.C when '0' is Constrained UNPREDICTABLE
// on the Tagged attribute
if (HaveMTE2Ext() && walkstate.memattrs.tags == MemTag_AllocationTagged &&
    !ConstrainUnpredictableBool(Unpredictable_S1CTAGGED)) then
    memattrs.tags = MemTag_Untagged;
else
    memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    memattrs.shareability = walkstate.memattrs.shareability;
else
    memattrs.shareability = EffectiveShareability(memattrs);

if accdesc.ls64 && memattrs.memtype == MemType_Normal then
    if memattrs.inner.attrs != MemAttr_NC || memattrs.outer.attrs != MemAttr_NC then
        fault.statuscode = Fault_Exclusive;
        return (fault, AddressDescriptor UNKNOWN);

ipa = CreateAddressDescriptor(va, oa, memattrs);
ipa.slassured = walkstate.slassured;
varange = AArch64.GetVARange(va);
ipa.mecid = AArch64.S1OutputMECID(walkparams, regime, varange, ipa.paddress.paspace,
                                descriptor);
return (fault, ipa);

```



```

// AArch64.S2Translate()
// =====
// Translate stage 1 IPA to PA and combine memory attributes

(FaultRecord, AddressDescriptor) AArch64.S2Translate(FaultRecord fault_in, AddressDescriptor ipa,
                                                    boolean slaarch64, boolean aligned,
                                                    AccessDescriptor accdesc)

walkparams = AArch64.GetS2TTWParams(accdesc.ss, ipa.paddress.paspace, slaarch64);
FaultRecord fault = fault_in;
boolean s2fslmro;

// Prepare fault fields in case a fault is detected
fault.statuscode = Fault\_None; // Ignore any faults from stage 1
fault.secondstage = TRUE;
fault.s2fslwalk = accdesc.acctype == AccessType\_TTW;
fault.ipaddress = ipa.paddress;

if walkparams.vm != '1' then
    // Stage 2 translation is disabled
    return (fault, ipa);

constant integer s2mintxsz = AArch64.S2MinTxSZ(walkparams.d128, walkparams.ds,
                                           walkparams.tgx, slaarch64);
constant integer s2maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
if AArch64.S2TxSZFaults(walkparams, slaarch64) then
    fault.statuscode = Fault\_Translation;
    fault.level = 0;
    return (fault, AddressDescriptor UNKNOWN);
elseif UInt(walkparams.txsz) < s2mintxsz then
    walkparams.txsz = s2mintxsz<5:0>;
elseif UInt(walkparams.txsz) > s2maxtxsz then
    walkparams.txsz = s2maxtxsz<5:0>;

if (walkparams.d128 == '0' &&
    (AArch64.S2InvalidSL(walkparams) || AArch64.S2InconsistentSL(walkparams))) then
    fault.statuscode = Fault\_Translation;
    fault.level = 0;
    return (fault, AddressDescriptor UNKNOWN);

if AArch64.IPAIsOutOfRange(ipa.paddress.address, walkparams) then
    fault.statuscode = Fault\_Translation;
    fault.level = 0;
    return (fault, AddressDescriptor UNKNOWN);

AddressDescriptor descpaddr;
TTWState walkstate;
bits(128) descriptor;
bits(128) new_desc;
bits(128) mem_desc;
repeat
    if walkparams.d128 == '1' then
        (fault, descpaddr, walkstate, descriptor) = AArch64.S2Walk(fault, ipa, walkparams,
                                                                accdesc, 128);
    else
        (fault, descpaddr, walkstate, descriptor<63:0>) = AArch64.S2Walk(fault, ipa,
                                                                walkparams, accdesc,
                                                                64);

        descriptor<127:64> = Zeros(64);
    if fault.statuscode != Fault\_None then
        return (fault, AddressDescriptor UNKNOWN);

    if AArch64.S2HasAlignmentFault(accdesc, aligned, walkstate.memattrs) then
        fault.statuscode = Fault\_Alignment;

    if fault.statuscode == Fault\_None then
        (fault, s2fslmro) = AArch64.S2CheckPermissions(fault, walkstate, walkparams, ipa,
                                                                accdesc);

    new_desc = descriptor;

```

```

if walkparams.ha == '1' && AArch64.SettingAccessFlagPermitted(fault) then
    // Set descriptor AF bit
    new_desc<10> = '1';

// If HW update of dirty bit is enabled, the walk state permissions
// will already reflect a configuration permitting writes.
// The update of the descriptor occurs only if the descriptor bits in
// memory do not reflect that and the access instigates a write.

if (AArch64.SettingDirtyStatePermitted(fault) &&
    walkparams.ha == '1' &&
    walkparams.hd == '1' &&
    (walkparams.s2pie == '1' || descriptor<51> == '1') &&
    accdesc.write &&
    !(accdesc.acctype IN {AccessType\_AT, AccessType\_IC, AccessType\_DC})) then
    // Set descriptor S2AP[1]/Dirty bit permitting stage 2 writes
    new_desc<7> = '1';

// Either the access flag was clear or S2AP[1]/Dirty is clear
if new_desc != descriptor then
    AccessDescriptor descaccess = CreateAccDescTTEUpdate(accdesc);
    if walkparams.dl28 == '1' then
        (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
                                                    walkparams.ee, descaccess,
                                                    descaddr);
    else
        (fault, mem_desc<63:0>) = AArch64.MemSwapTableDesc(fault, descriptor<63:0>,
                                                            new_desc<63:0>, walkparams.ee,
                                                            descaccess, descaddr);

        mem_desc<127:64> = Zeros(64);

until new_desc == descriptor || mem_desc == new_desc;

if fault.statuscode != Fault\_None then
    return (fault, AddressDescriptor UNKNOWN);

ipa_64 = ZeroExtend(ipa.paddress.address, 64);
// Output Address
oa = StageOA(ipa_64, walkparams.dl28, walkparams.tgx, walkstate);
MemoryAttributes s2_memattrs;
if (accdesc.acctype == AccessType\_TTW &&
    walkstate.memattrs.memtype == MemType\_Device && walkparams.ptw == '0') ||
    (accdesc.acctype == AccessType\_IFETCH &&
    (walkstate.memattrs.memtype == MemType\_Device || HCR_EL2.ID == '1')) ||
    (accdesc.acctype != AccessType\_IFETCH &&
    walkstate.memattrs.memtype == MemType\_Normal && !S2DCacheEnabled()) then
    // Treat memory attributes as Normal Non-Cacheable
    s2_memattrs = NormalNCMemAttr();
    s2_memattrs.xs = walkstate.memattrs.xs;
else
    s2_memattrs = walkstate.memattrs;

if accdesc.ls64 && s2_memattrs.memtype == MemType\_Normal then
    if s2_memattrs.inner.attrs != MemAttr\_NC || s2_memattrs.outer.attrs != MemAttr\_NC then
        fault.statuscode = Fault\_Exclusive;
        return (fault, AddressDescriptor UNKNOWN);

s2aarch64 = TRUE;
MemoryAttributes memattrs;
if walkparams.fwb == '0' then
    memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs, s2aarch64);
else
    memattrs = s2_memattrs;

pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs);
pa.s2fslmro = s2fslmro;
pa.mecid = AArch64.S2OutputMECID(walkparams, pa.paddress.paspace, descriptor);
return (fault, pa);

```

## Library pseudocode for aarch64/translation/vmsa\_translation/AArch64.SettingAccessFlagPermitted

```
// AArch64.SettingAccessFlagPermitted()
// =====
// Determine whether the access flag could be set by HW given the fault status

boolean AArch64.SettingAccessFlagPermitted(FaultRecord fault)
    if fault.statuscode == Fault\_None then
        return TRUE;
    elsif fault.statuscode IN {Fault\_Alignment, Fault\_Permission} then
        return ConstrainUnpredictableBool(Unpredictable\_AFUPDATE);
    else
        return FALSE;
```

## Library pseudocode for aarch64/translation/vmsa\_translation/AArch64.SettingDirtyStatePermitted

```
// AArch64.SettingDirtyStatePermitted()
// =====
// Determine whether the dirty state could be set by HW given the fault status

boolean AArch64.SettingDirtyStatePermitted(FaultRecord fault)
    if fault.statuscode == Fault\_None then
        return TRUE;
    elsif fault.statuscode == Fault\_Alignment then
        return ConstrainUnpredictableBool(Unpredictable\_DBUPDATE);
    else
        return FALSE;
```

## Library pseudocode for aarch64/translation/vmsa\_translation/AArch64.TranslateAddress

```
// AArch64.TranslateAddress()
// =====
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) va, AccessDescriptor accdesc,
                                           boolean aligned, integer size)
    if (SPESampleInFlight && !(accdesc.acctype IN {AccessType\_IFETCH,
                                                  AccessType\_SPE})) then
        SPEStartCounter(SPECounterPosTranslationLatency);

    AddressDescriptor result = AArch64.FullTranslate(va, accdesc, aligned);

    if !IsFault(result) && accdesc.acctype != AccessType\_IFETCH then
        result.fault = AArch64.CheckDebug(va, accdesc, size);

    if HaveRME() && !IsFault(result) && (
        accdesc.acctype != AccessType\_DC ||
        boolean IMPLEMENTATION_DEFINED "GPC Fault on DC operations") then
        result.fault.gpcf = GranuleProtectionCheck(result, accdesc);

        if result.fault.gpcf.gpcf != GPCF\_None then
            result.fault.statuscode = Fault\_GPCFOnOutput;
            result.fault.paddress = result.paddress;

    if !IsFault(result) && accdesc.acctype == AccessType\_IFETCH then
        result.fault = AArch64.CheckDebug(va, accdesc, size);

    if (SPESampleInFlight && !(accdesc.acctype IN {AccessType\_IFETCH,
                                                  AccessType\_SPE})) then
        SPEStopCounter(SPECounterPosTranslationLatency);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(va, 64);

    return result;
```



## Library pseudocode for aarch64/translation/vmsa\_ttentry/AArch64.BlockDescSupported

```
// AArch64.BlockDescSupported()
// =====
// Determine whether a block descriptor is valid for the given granule size
// and level

boolean AArch64.BlockDescSupported(bit d128, bit ds, TGx tgx, integer level)
    case tgx of
        when TGx\_4KB return ((level == 0 && (ds == '1' || d128 == '1')) ||
                                level == 1 ||
                                level == 2);
        when TGx\_16KB return ((level == 1 && (ds == '1' || d128 == '1')) ||
                                level == 2);
        when TGx\_64KB return ((level == 1 && (d128 == '1' || AArch64.PAMax() >= 52)) ||
                                level == 2);
    return FALSE;
```

## Library pseudocode for aarch64/translation/vmsa\_ttentry/AArch64.BlocknTFaults

```
// AArch64.BlocknTFaults()
// =====
// Identify whether the nT bit in a block descriptor is effectively set
// causing a translation fault

boolean AArch64.BlocknTFaults(bit d128, bits(N) descriptor)
    bit nT;
    if !HaveBlockBBM() then
        return FALSE;
    nT = if d128 == '1' then descriptor<6> else descriptor<16>;
    bbm_level = AArch64.BlockBBMSupportLevel();
    nT_faults = (boolean IMPLEMENTATION_DEFINED
                 "BBM level 1 or 2 support nT bit causes Translation Fault");

    return bbm_level IN {1, 2} && nT == '1' && nT_faults;
```

## Library pseudocode for aarch64/translation/vmsa\_ttentry/AArch64.ContiguousBit

```
// AArch64.ContiguousBit()
// =====
// Get the value of the contiguous bit

bit AArch64.ContiguousBit(TGx tgx, bit d128, integer level, bits(N) descriptor)
    if d128 == '1' then
        if (tgx == TGx\_64KB && level == 1) || (tgx == TGx\_4KB && level == 0) then
            return '0'; // RES0
        else
            return descriptor<111>;
    // When using TGx 64KB and FEAT_LPA is implemented,
    // the Contiguous bit is RES0 for Block descriptors at level 1

    if tgx == TGx\_64KB && level == 1 then
        return '0'; // RES0

    // When the effective value of TCR_ELx.DS is '1',
    // the Contiguous bit is RES0 for all the following:
    // * For TGx 4KB, Block descriptors at level 0
    // * For TGx 16KB, Block descriptors at level 1

    if tgx == TGx\_16KB && level == 1 then
        return '0'; // RES0

    if tgx == TGx\_4KB && level == 0 then
        return '0'; // RES0

    return descriptor<52>;
```

## Library pseudocode for aarch64/translation/vmsa\_ttentry/AArch64.DecodeDescriptorType

```
// AArch64.DecodeDescriptorType()
// =====
// Determine whether the descriptor is a page, block or table

DescriptorType AArch64.DecodeDescriptorType(bits(N) descriptor, bit d128, bit ds,
                                           TGx tgx, integer level)

    if descriptor<0> == '0' then
        return DescriptorType\_Invalid;
    elsif d128 == '1' then
        bits(2) skl = descriptor<110:109>;
        if tgx IN {TGx 16KB, TGx 64KB} && UInt(skl) == 3 then
            return DescriptorType\_Invalid;

            integer effective_level = level + UInt(skl);
            if effective_level > FINAL\_LEVEL then
                return DescriptorType\_Invalid;
            elsif effective_level == FINAL\_LEVEL then
                return DescriptorType\_Leaf;
            else
                return DescriptorType\_Table;
        else
            if descriptor<1> == '1' then
                if level == FINAL\_LEVEL then
                    return DescriptorType\_Leaf;
                else
                    return DescriptorType\_Table;
            elsif descriptor<1> == '0' then
                if AArch64.BlockDescSupported(d128, ds, tgx, level) then
                    return DescriptorType\_Leaf;
                else
                    return DescriptorType\_Invalid;
```

## Library pseudocode for aarch64/translation/vmsa\_ttentry/AArch64.S1ApplyOutputPerms

```
// AArch64.S1ApplyOutputPerms()
// =====
// Apply output permissions encoded in stage 1 page/block descriptors

Permissions AArch64.S1ApplyOutputPerms(Permissions permissions_in, bits(N) descriptor,
                                         Regime regime, S1TTWParams walkparams)
    Permissions permissions = permissions_in;

    bits (4) pi_index;
    if walkparams.pie == '1' then
        if walkparams.dl28 == '1' then
            pi_index = descriptor<118:115>;
        else
            pi_index = descriptor<54:53>:descriptor<51>:descriptor<6>;
        bit_index      = 4 * UInt(pi_index);
        permissions.ppi = walkparams.pir<bit_index+3:bit_index>;
        permissions.upi = walkparams.pire0<bit_index+3:bit_index>;
        permissions.ndirty = descriptor<7>;
    else
        if regime == Regime\_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
            permissions.ap<2:1> = descriptor<7>:'0';
            permissions.pxn      = descriptor<54>;
        elsif HasUnprivileged(regime) then
            permissions.ap<2:1> = descriptor<7:6>;
            permissions.uxn      = descriptor<54>;
            permissions.pxn      = descriptor<53>;
        else
            permissions.ap<2:1> = descriptor<7>:'1';
            permissions.xn      = descriptor<54>;
        // Descriptors marked with DBM set have the effective value of AP[2] cleared.
        // This implies no Permission faults caused by lack of write permissions are
        // reported, and the Dirty bit can be set.
        if walkparams.ha == '1' && walkparams.hd == '1' && descriptor<51> == '1' then
            permissions.ap<2> = '0';

    if IsFeatureImplemented(FEAT_S1POE) then
        if walkparams.dl28 == '1' then
            permissions.po_index = descriptor<124:121>;
        else
            permissions.po_index = '0':descriptor<62:60>;

    return permissions;
```

## Library pseudocode for aarch64/translation/vmsa\_ttentry/AArch64.S1ApplyTablePerms

```
// AArch64.S1ApplyTablePerms()
// =====
// Apply hierarchical permissions encoded in stage 1 table descriptors

Permissions AArch64.S1ApplyTablePerms(Permissions permissions_in, bits(N) descriptor,
                                       Regime regime, S1TTWParams walkparams)
    Permissions permissions = permissions_in;
    bits(2) ap_table;
    bit pxn_table;
    bit uxn_table;
    bit xn_table;
    if regime == Regime_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
        if walkparams.d128 == '1' then
            ap_table = descriptor<126>:'0';
            pxn_table = descriptor<124>;
        else
            ap_table = descriptor<62>:'0';
            pxn_table = descriptor<60>;
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;

    elseif HasUnprivileged(regime) then
        if walkparams.d128 == '1' then
            ap_table = descriptor<126:125>;
            uxn_table = descriptor<124>;
            pxn_table = descriptor<123>;
        else
            ap_table = descriptor<62:61>;
            uxn_table = descriptor<60>;
            pxn_table = descriptor<59>;
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.uxn_table = permissions.uxn_table OR uxn_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;

    else
        if walkparams.d128 == '1' then
            ap_table = descriptor<126>:'0';
            xn_table = descriptor<124>;
        else
            ap_table = descriptor<62>:'0';
            xn_table = descriptor<60>;
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.xn_table = permissions.xn_table OR xn_table;

    return permissions;
```

## Library pseudocode for aarch64/translation/vmsa\_ttentry/AArch64.S2ApplyOutputPerms

```
// AArch64.S2ApplyOutputPerms()
// =====
// Apply output permissions encoded in stage 2 page/block descriptors

Permissions AArch64.S2ApplyOutputPerms(bits(N) descriptor, S2TTWParams walkparams)
    Permissions permissions;
    bits(4) s2pi_index;
    if walkparams.s2pie == '1' then
        if walkparams.dl28 == '1' then
            s2pi_index = descriptor<118:115>;
        else
            s2pi_index = descriptor<54:53,51,6>;
        bit_index = 4 * UInt(s2pi_index);
        permissions.s2pi = walkparams.s2pir<bit_index+3 : bit_index>;
        permissions.s2dirty = descriptor<7>;
    else
        permissions.s2ap = descriptor<7:6>;
        if walkparams.dl28 == '1' then
            permissions.s2xn = descriptor<118>;
        else
            permissions.s2xn = descriptor<54>;

        if HaveExtendedExecuteNeverExt() then
            if walkparams.dl28 == '1' then
                permissions.s2xnx = descriptor<117>;
            else
                permissions.s2xnx = descriptor<53>;
        else
            permissions.s2xnx = '0';

        // Descriptors marked with DBM set have the effective value of S2AP[1] set.
        // This implies no Permission faults caused by lack of write permissions are
        // reported, and the Dirty bit can be set.
        bit desc_dbm;
        if walkparams.dl28 == '1' then
            desc_dbm = descriptor<115>;
        else
            desc_dbm = descriptor<51>;
        if walkparams.ha == '1' && walkparams.hd == '1' && desc_dbm == '1' then
            permissions.s2ap<1> = '1';
    if IsFeatureImplemented(FEAT_S2POE) then
        if walkparams.dl28 == '1' then
            permissions.s2po_index = descriptor<124:121>;
        else
            permissions.s2po_index = descriptor<62:59>;
    return permissions;
```

## Library pseudocode for aarch64/translation/vmsa\_walk/AArch64.S1InitialTTWState

```
// AArch64.S1InitialTTWState()
// =====
// Set properties of first access to translation tables in stage 1

TTWState AArch64.S1InitialTTWState(S1TTWParams walkparams, bits(64) va, Regime regime,
                                   SecurityState ss)

    TTWState      walkstate;
    FullAddress   tablebase;
    Permissions   permissions;
    bits(128)     ttbr;

    ttbr          = AArch64.S1TTBR(regime, va);
    case ss of
        when SS Secure      tablebase.paspace = PAS Secure;
        when SS NonSecure   tablebase.paspace = PAS NonSecure;
        when SS Root        tablebase.paspace = PAS Root;
        when SS Realm       tablebase.paspace = PAS Realm;

    tablebase.address = AArch64.S1TTBaseAddress(walkparams, regime, ttbr);

    permissions.ap_table = '00';
    if HasUnprivileged(regime) then
        permissions.uxn_table = '0';
        permissions.pxn_table = '0';
    else
        permissions.xn_table = '0';

    walkstate.baseaddress = tablebase;
    walkstate.level       = AArch64.S1StartLevel(walkparams);
    walkstate.istable     = TRUE;
    // In regimes that support global and non-global translations, translation
    // table entries from lookup levels other than the final level of lookup
    // are treated as being non-global
    walkstate.nG          = if HasUnprivileged(regime) then '1' else '0';
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
    walkstate.permissions = permissions;
    if (regime == Regime\_EL10 && EL2Enabled() && (HCR_EL2.VM == '1' || HCR_EL2.DC == '1')) then
        if ((AArch64.GetVARange(va) == VARange\_LOWER && VTCR_EL2.TL0 == '1') ||
            (AArch64.GetVARange(va) == VARange\_UPPER && VTCR_EL2.TL1 == '1')) then
            walkstate.slassured = TRUE;
        else
            walkstate.slassured = FALSE;
    else
        walkstate.slassured = FALSE;
    walkstate.disch = walkparams.disch;

    return walkstate;
```



```

// AArch64.S1NextWalkStateLeaf()
// =====
// Decode stage 1 page or block descriptor as output to this stage of translation

TTWState AArch64.S1NextWalkStateLeaf(TTWState currentstate, boolean s2fslmro, Regime regime,
                                     SecurityState ss, S1TWPParams walkparams, bits(N) descriptor)

    TTWState    nextstate;
    FullAddress baseaddress;
    baseaddress.address = AArch64.LeafBase(descriptor, walkparams.dl28,
                                         walkparams.ds,
                                         walkparams.tgx, currentstate.level);

    if currentstate.baseaddress.paspace == PAS\_Secure then
        // Determine PA space of the block from NS bit
        bit ns;
        ns = if walkparams.dl28 == '1' then descriptor<127> else descriptor<5>;
        baseaddress.paspace = if ns == '0' then PAS\_Secure else PAS\_NonSecure;
    elseif currentstate.baseaddress.paspace == PAS\_Root then
        // Determine PA space of the block from NSE and NS bits
        bit nse;
        bit ns;
        <nse,ns> = if walkparams.dl28 == '1' then descriptor<11,127> else descriptor<11,5>;
        baseaddress.paspace = DecodePASpace(nse, ns);

        // If Secure state is not implemented, but RME is,
        // force Secure space accesses to Non-secure space
        if baseaddress.paspace == PAS\_Secure && !HaveSecureState() then
            baseaddress.paspace = PAS\_NonSecure;

    elseif (currentstate.baseaddress.paspace == PAS\_Realm &&
           regime IN {Regime\_EL2, Regime\_EL20}) then
        // Realm EL2 and EL2&0 regimes have a stage 1 NS bit
        bit ns;
        ns = if walkparams.dl28 == '1' then descriptor<127> else descriptor<5>;
        baseaddress.paspace = if ns == '0' then PAS\_Realm else PAS\_NonSecure;
    elseif currentstate.baseaddress.paspace == PAS\_Realm then
        // Realm EL1&0 regime does not have a stage 1 NS bit
        baseaddress.paspace = PAS\_Realm;
    else
        baseaddress.paspace = PAS\_NonSecure;

    nextstate.istable      = FALSE;
    nextstate.level       = currentstate.level;
    nextstate.baseaddress = baseaddress;

    bits(4) attrindx;
    if walkparams.aie == '1' then
        if walkparams.dl28 == '1' then
            attrindx = descriptor<5:2>;
        else
            attrindx = descriptor<59,4:2>;
    else
        attrindx = '0':descriptor<4:2>;

    bits(2) sh;
    if walkparams.dl28 == '1' then
        sh = descriptor<9:8>;
    elseif walkparams.ds == '1' then
        sh = walkparams.sh;
    else
        sh = descriptor<9:8>;
    attr = AArch64.MAIRAttr(UInt(attrindx), walkparams.mair2, walkparams.mair);
    slaarch64 = TRUE;

    nextstate.memattrs     = S1DecodeMemAttrs(attr, sh, slaarch64, walkparams);
    nextstate.permissions = AArch64.S1ApplyOutputPerms(currentstate.permissions,
                                                         descriptor, regime, walkparams);

    bit protectedbit;
    if walkparams.dl28 == '1' then
        protectedbit = descriptor<114>;

```



```

else
    protectedbit = if walkparams.pnch == '1' then descriptor<52> else '0';
if (currentstate.slassured && s2fslmro && protectedbit == '1') then
    nextstate.slassured = TRUE;
else
    nextstate.slassured = FALSE;

if walkparams.pnch == '1' || currentstate.disch == '1' then
    nextstate.contiguous = '0';
else
    nextstate.contiguous = AArch64.ContiguousBit(walkparams.tgx, walkparams.d128,
                                                currentstate.level, descriptor);
if !HasUnprivileged(regime) then
    nextstate.nG = '0';
elseif ss == SS\_Secure && currentstate.baseaddress.paspace == PAS\_NonSecure then
    // In Secure state, a translation must be treated as non-global,
    // regardless of the value of the nG bit,
    // if NSTable is set to 1 at any level of the translation table walk
    nextstate.nG = '1';
else
    nextstate.nG = descriptor<11>;

if walkparams.d128 == '1' then
    nextstate.guardedpage = descriptor<113>;
else
    nextstate.guardedpage = descriptor<50>;

return nextstate;

```

## Library pseudocode for aarch64/translation/vmsa\_walk/AArch64.S1NextWalkStateTable

```
// AArch64.S1NextWalkStateTable()
// =====
// Decode stage 1 table descriptor to transition to the next level

TTWState AArch64.S1NextWalkStateTable(TTWState currentstate, boolean s2fslmro, Regime regime,
S1TTWParams walkparams, bits(N) descriptor)

TTWState    nextstate;
FullAddress tablebase;
bits(2) skl = if walkparams.d128 == '1' then descriptor<110:109> else '00';

tablebase.address = AArch64.NextTableBase(descriptor, walkparams.d128,
                                           skl, walkparams.ds,
                                           walkparams.tgx);

if currentstate.baseaddress.paspace == PAS\_Secure then
    // Determine PA space of the next table from NSTable bit
    bit nstable;
    nstable = if walkparams.d128 == '1' then descriptor<127> else descriptor<63>;
    tablebase.paspace = if nstable == '0' then PAS\_Secure else PAS\_NonSecure;
else
    // Otherwise bit 63 is RES0 and there is no NSTable bit
    tablebase.paspace = currentstate.baseaddress.paspace;

nextstate.istable      = TRUE;
nextstate.nG          = currentstate.nG;
if walkparams.d128 == '1' then
    nextstate.level    = currentstate.level + UInt(skl) + 1;
else
    nextstate.level    = currentstate.level + 1;
nextstate.baseaddress = tablebase;
nextstate.memattrrs   = currentstate.memattrrs;
if walkparams.hpd == '0' && walkparams.pie == '0' then
    nextstate.permissions = AArch64.S1ApplyTablePerms(currentstate.permissions, descriptor,
                                                         regime, walkparams);
else
    nextstate.permissions = currentstate.permissions;
bit protectedbit;
if walkparams.d128 == '1' then
    protectedbit = descriptor<114>;
else
    protectedbit = if walkparams.pnch == '1' then descriptor<52> else '0';
if (currentstate.slassured && s2fslmro && protectedbit == '1') then
    nextstate.slassured = TRUE;
else
    nextstate.slassured = FALSE;
nextstate.disch = if walkparams.d128 == '1' then descriptor<112> else '0';

return nextstate;
```



```

// AArch64.S1Walk()
// =====
// Traverse stage 1 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

(FaultRecord, AddressDescriptor, TTWState, bits(N)) AArch64.S1Walk(FaultRecord fault_in,
                                                                    S1TTWParams walkparams,
                                                                    bits(64) va, Regime regime,
                                                                    AccessDescriptor accdesc,
                                                                    integer N)

FaultRecord fault = fault_in;
boolean slaarch64;
boolean aligned;

if HasUnprivileged(regime) && AArch64.S1EPD(regime, va) == '1' then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
            bits(N) UNKNOWN);

walkstate = AArch64.S1InitialTTWState(walkparams, va, regime, accdesc.ss);
constant integer startlevel = walkstate.level;

bits(N) descriptor;
AddressDescriptor walkaddress;
bits(2) skl = '00';
walkaddress.vaddress = va;
walkaddress.mecid = AArch64.TTWalkMECID(walkparams.emec, regime, accdesc.ss);

if !AArch64.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime\_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

boolean s2fslmro = FALSE;

DescriptorType descstype;
FullAddress descaddress = AArch64.S1SLTTEEntryAddress(walkstate.level, walkparams, va,
                                                         walkstate.baseaddress);

// Detect Address Size Fault by Descriptor Address
if AArch64.OAOutOfRange(descaddress.address, walkparams.d128,
                        walkparams.ps, walkparams.tgx) then
    fault.statuscode = Fault\_AddressSize;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

repeat
    fault.level = walkstate.level;
    walkaddress.paddress = descaddress;
    walkaddress.slassured = walkstate.slassured;

    boolean toplevel = walkstate.level == startlevel;
    VARange varange = AArch64.GetVARange(va);
    AccessDescriptor walkaccess = CreateAccDescS1TTW(toplevel, varange, accdesc);
    FaultRecord s2fault;
    AddressDescriptor s2walkaddress;
    if regime == Regime\_EL10 && EL2Enabled() then
        slaarch64 = TRUE;
        aligned = TRUE;
        (s2fault, s2walkaddress) = AArch64.S2Translate(fault, walkaddress, slaarch64, aligned,

```

```

walkaccess);

if s2fault.statuscode != Fault\_None then
    return (s2fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
            bits(N) UNKNOWN);

s2fslmro = s2walkaddress.s2fslmro;
(fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, walkaccess,
                                       fault, N);
else
    (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, walkaccess,
                                       fault, N);

if fault.statuscode != Fault\_None then
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
            bits(N) UNKNOWN);

bits(N) new_descriptor;
repeat
    new_descriptor = descriptor;
    descstype = AArch64.DecodeDescriptorType(descriptor, walkparams.d128, walkparams.ds,
                                             walkparams.tgx, walkstate.level);
    case descstype of
        when DescriptorType\_Table
            walkstate = AArch64.S1NextWalkStateTable(walkstate, s2fslmro,
                                                    regime, walkparams, descriptor);
            skl = if walkparams.d128 == '1' then descriptor<110:109> else '00';
            descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.d128, skl,
                                                walkparams.tgx, walkparams.txsz, va,
                                                walkstate.baseaddress);

            // Detect Address Size Fault by Descriptor Address
            if AArch64.OAOutOfRange(descaddress.address, walkparams.d128,
                                    walkparams.ps, walkparams.tgx) then
                fault.statuscode = Fault\_AddressSize;
                return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                        bits(N) UNKNOWN);

            if walkparams.haft == '1' then
                new_descriptor<10> = '1';
            if (walkparams.d128 == '1' && skl != '00' &&
                AArch64.BlocknTFaults(walkparams.d128, descriptor)) then
                fault.statuscode = Fault\_Translation;
                return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                        bits(N) UNKNOWN);
        when DescriptorType\_Leaf
            walkstate = AArch64.S1NextWalkStateLeaf(walkstate, s2fslmro,
                                                    regime, accdesc.ss, walkparams,
                                                    descriptor);
        when DescriptorType\_Invalid
            fault.statuscode = Fault\_Translation;
            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                    bits(N) UNKNOWN);
    otherwise
        Unreachable();

if new_descriptor != descriptor then
    AddressDescriptor descpaddr;
    AccessDescriptor descaccess = CreateAccDescTTEUpdate(accdesc);
    if regime == Regime\_EL10 && EL2Enabled() then
        slaarch64 = TRUE;
        aligned = TRUE;
        (s2fault, descpaddr) = AArch64.S2Translate(fault, walkaddress,
                                                  slaarch64, aligned,
                                                  descaccess);

        if s2fault.statuscode != Fault\_None then
            return (s2fault, AddressDescriptor UNKNOWN,
                    TTWState UNKNOWN, bits(N) UNKNOWN);
    else

```

```

        descaddr = walkaddress;

        (fault, descriptor) = AArch64.MemSwapTableDesc(fault, descriptor, new_descriptor,
                                                    walkparams.ee, descaccess,
                                                    descaddr);

        if fault.statuscode != Fault\_None then
            return (fault, AddressDescriptor UNKNOWN,
                    TTWState UNKNOWN, bits(N) UNKNOWN);
        until new_descriptor == descriptor;
    until descype == DescriptorType\_Leaf;

    FullAddress oa = StageOA(va, walkparams.dl28, walkparams.tgx, walkstate);

    if (walkstate.contiguous == '1' &&
        AArch64.ContiguousBitFaults(walkparams.dl28, walkparams.txs, walkparams.tgx,
                                     walkstate.level)) then
        fault.statuscode = Fault\_Translation;
    elsif (descype == DescriptorType\_Leaf && walkstate.level < FINAL\_LEVEL &&
          AArch64.BlocknTFaults(walkparams.dl28, descriptor)) then
        fault.statuscode = Fault\_Translation;
    elsif AArch64.SlAMECFault(walkparams, walkstate.baseaddress.paspace, regime, descriptor) then
        fault.statuscode = Fault\_Translation;
    // Detect Address Size Fault by final output
    elsif AArch64.OAOutOfRange(oa.address, walkparams.dl28,
                               walkparams.ps, walkparams.tgx) then
        fault.statuscode = Fault\_AddressSize;
    // Check descriptor AF bit
    elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
          !(accdesc.acctype IN {AccessType\_DC, AccessType\_IC} &&
          !boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations")) then
        fault.statuscode = Fault\_AccessFlag;

    if fault.statuscode != Fault\_None then
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

    return (fault, walkaddress, walkstate, descriptor);

```

### Library pseudocode for aarch64/translation/vmsa\_walk/AArch64.S2InitialTTWState

```

// AArch64.S2InitialTTWState()
// =====
// Set properties of first access to translation tables in stage 2

TTWState AArch64.S2InitialTTWState(SecurityState ss, S2TTWParams walkparams)
    TTWState walkstate;
    FullAddress tablebase;
    bits(128) ttbr;

    ttbr = ZeroExtend(VTTBR_EL2, 128);
    case ss of
        when SS\_NonSecure tablebase.paspace = PAS\_NonSecure;
        when SS\_Realm tablebase.paspace = PAS\_Realm;
    tablebase.address = AArch64.S2TTBaseAddress(walkparams, tablebase.paspace, ttbr);

    walkstate.baseaddress = tablebase;
    walkstate.level = AArch64.S2StartLevel(walkparams);
    walkstate.istable = TRUE;
    walkstate.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);

    return walkstate;

```

## Library pseudocode for aarch64/translation/vmsa\_walk/AArch64.S2NextWalkStateLeaf

```
// AArch64.S2NextWalkStateLeaf()
// =====
// Decode stage 2 page or block descriptor as output to this stage of translation

TTWState AArch64.S2NextWalkStateLeaf(TTWState currentstate, SecurityState ss,
                                       S2TTWParams walkparams, AddressDescriptor ipa,
                                       bits(N) descriptor)

TTWState    nextstate;
FullAddress baseaddress;

if ss == SS\_Secure then
    baseaddress.paspace = AArch64.SS2OutputPASpace(walkparams, ipa.paddress.paspace);
elsif ss == SS\_Realm then
    bit ns;
    ns = if walkparams.d128 == '1' then descriptor<127> else descriptor<55>;
    baseaddress.paspace = if ns == '1' then PAS\_NonSecure else PAS\_Realm;
else
    baseaddress.paspace = PAS\_NonSecure;
baseaddress.address = AArch64.LeafBase(descriptor, walkparams.d128, walkparams.ds,
                                       walkparams.tgx, currentstate.level);

nextstate.istable      = FALSE;
nextstate.level        = currentstate.level;
nextstate.baseaddress = baseaddress;
nextstate.permissions = AArch64.S2ApplyOutputPerms(descriptor, walkparams);

s2_attr = descriptor<5:2>;
s2_sh   = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
s2_fnxs = descriptor<11>;
if walkparams.fwb == '1' then
    nextstate.memattrs = AArch64.S2ApplyFWBMemAttrs(ipa.memattrs, walkparams, descriptor);
    if s2_attr<3:1> == '111' then
        nextstate.permissions.s2tag_na = '1';
    else
        nextstate.permissions.s2tag_na = '0';
else
    s2aarch64 = TRUE;
    nextstate.memattrs = S2DecodeMemAttrs(s2_attr, s2_sh, s2aarch64);
    // FnXS is used later to mask the XS value from stage 1
    nextstate.memattrs.xs = NOT s2_fnxs;
    if s2_attr == '0100' then
        nextstate.permissions.s2tag_na = '1';
    else
        nextstate.permissions.s2tag_na = '0';
nextstate.contiguous = AArch64.ContiguousBit(walkparams.tgx, walkparams.d128,
                                             currentstate.level, descriptor);

if walkparams.d128 == '1' then
    nextstate.s2assuredonly = descriptor<114>;
else
    nextstate.s2assuredonly = if walkparams.assuredonly == '1' then descriptor<58> else '0';

return nextstate;
```

## Library pseudocode for aarch64/translation/vmsa\_walk/AArch64.S2NextWalkStateTable

```
// AArch64.S2NextWalkStateTable()
// =====
// Decode stage 2 table descriptor to transition to the next level

TTWState AArch64.S2NextWalkStateTable(TTWState currentstate, S2TTWParams walkparams,
                                       bits(N) descriptor)

    TTWState    nextstate;
    FullAddress tablebase;
    bits(2) skl = if walkparams.d128 == '1' then descriptor<110:109> else '00';

    tablebase.address = AArch64.NextTableBase(descriptor, walkparams.d128,
                                              skl, walkparams.ds,
                                              walkparams.tgx);
    tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable      = TRUE;
    if walkparams.d128 == '1' then
        nextstate.level    = currentstate.level + UInt(skl) + 1;
    else
        nextstate.level    = currentstate.level + 1;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs    = currentstate.memattrs;

    return nextstate;
```





```

// AArch64.S2Walk()
// =====
// Traverse stage 2 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

(FaultRecord, AddressDescriptor, TTWState, bits(N)) AArch64.S2Walk(FaultRecord fault_in,
                                                                    AddressDescriptor ipa,
                                                                    S2TTWParams walkparams,
                                                                    AccessDescriptor accdesc,
                                                                    integer N)

FaultRecord fault = fault_in;
ipa_64 = ZeroExtend(ipa.paddress.address, 64);

TTWState walkstate;
if accdesc.ss == SS\_Secure then
    walkstate = AArch64.SS2InitialTTWState(walkparams, ipa.paddress.paspace);
else
    walkstate = AArch64.S2InitialTTWState(accdesc.ss, walkparams);

constant integer startlevel = walkstate.level;

bits(N) descriptor;
AccessDescriptor walkaccess = CreateAccDescS2TTW(accdesc);
AddressDescriptor walkaddress;
bits(2) skl = '00';

walkaddress.vaddress = ipa.vaddress;
walkaddress.mecid = AArch64.TTWalkMECID(walkparams.emec, Regime\_EL10, accdesc.ss);

if !S2DCacheEnabled() then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

DescriptorType desctype;

// Initial lookup might index into concatenated tables
FullAddress descaddress = AArch64.S2SLTTEnterAddress(walkparams, ipa.paddress.address,
                                                    walkstate.baseaddress);

// Detect Address Size Fault by Descriptor Address
if AArch64.OAOutOfRange(descaddress.address, walkparams.d128, walkparams.ps,
                        walkparams.tgx) then
    fault.statuscode = Fault\_AddressSize;
    fault.level = 0;
    return (fault, AccessDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

repeat
    fault.level = walkstate.level;
    walkaddress.paddress = descaddress;
    (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, walkaccess, fault, N);

    if fault.statuscode != Fault\_None then
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

    bits(N) new_descriptor;
    repeat
        new_descriptor = descriptor;
        desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.d128, walkparams.ds,
                                                walkparams.tgx, walkstate.level);

        case desctype of
            when DescriptorType\_Table
                walkstate = AArch64.S2NextWalkStateTable(walkstate, walkparams, descriptor);
                skl = if walkparams.d128 == '1' then descriptor<110:109> else '00';
                descaddress = AArch64.TTEnterAddress(walkstate.level, walkparams.d128, skl,
                                                    walkparams.tgx, walkparams.txsz, ipa_64,

```

```

                                walkstate.baseaddress);

// Detect Address Size Fault by table descriptor
if AArch64.OAOutOfRange(descaddress.address, walkparams.dl28, walkparams.ps,
                        walkparams.tgx) then
    fault.statuscode = Fault\_AddressSize;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
            bits(N) UNKNOWN);

if walkparams.haft == '1' then
    new_descriptor<10> = '1';

if (walkparams.dl28 == '1' && skl != '00' &&
    AArch64.BlocknTFaults(walkparams.dl28, descriptor)) then
    fault.statuscode = Fault\_Translation;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
            bits(N) UNKNOWN);

when DescriptorType\_Leaf
    walkstate = AArch64.S2NextWalkStateLeaf(walkstate, accdesc.ss, walkparams, ipa,
                                            descriptor);

when DescriptorType\_Invalid
    fault.statuscode = Fault\_Translation;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

otherwise
    Unreachable();

if new_descriptor != descriptor then
    AccessDescriptor descaccess = CreateAccDescTTEUpdate(accdesc);
    (fault, descriptor) = AArch64.MemSwapTableDesc(fault, descriptor, new_descriptor,
                                                    walkparams.ee, descaccess,
                                                    walkaddress);

    if fault.statuscode != Fault\_None then
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);
    until new_descriptor == descriptor;
until desctype == DescriptorType\_Leaf;

FullAddress oa = StageOA(ipa_64, walkparams.dl28, walkparams.tgx, walkstate);

if (walkstate.contiguous == '1' &&
    AArch64.ContiguousBitFaults(walkparams.dl28, walkparams.txsz, walkparams.tgx,
                                walkstate.level)) then
    fault.statuscode = Fault\_Translation;
elseif (desctype == DescriptorType\_Leaf && walkstate.level < FINAL\_LEVEL &&
        AArch64.BlocknTFaults(walkparams.dl28, descriptor)) then
    fault.statuscode = Fault\_Translation;
// Detect Address Size Fault by final output
elseif AArch64.OAOutOfRange(oa.address, walkparams.dl28, walkparams.ps, walkparams.tgx) then
    fault.statuscode = Fault\_AddressSize;
// Check descriptor AF bit
elseif (descriptor<10> == '0' && walkparams.ha == '0' &&
        !(accdesc.acctype IN {AccessType\_DC, AccessType\_IC} &&
        !boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations")) then
    fault.statuscode = Fault\_AccessFlag;

return (fault, walkaddress, walkstate, descriptor);

```

## Library pseudocode for aarch64/translation/vmsa\_walk/AArch64.SS2InitialTTWState

```
// AArch64.SS2InitialTTWState()
// =====
// Set properties of first access to translation tables in Secure stage 2

TTWState AArch64.SS2InitialTTWState(S2TTWParams walkparams, PASpace ipaspace)
    TTWState walkstate;
    FullAddress tablebase;
    bits(128) ttbr;

    if ipaspace == PAS_Secure then
        ttbr = ZeroExtend(VSTTBR_EL2, 128);
    else
        ttbr = ZeroExtend(VTTBR_EL2, 128);

    if ipaspace == PAS_Secure then
        if walkparams.sw == '0' then
            tablebase.paspace = PAS_Secure;
        else
            tablebase.paspace = PAS_NonSecure;
    else
        if walkparams.nsw == '0' then
            tablebase.paspace = PAS_Secure;
        else
            tablebase.paspace = PAS_NonSecure;

    tablebase.address = AArch64.S2TTBaseAddress(walkparams, tablebase.paspace, ttbr);

    walkstate.baseaddress = tablebase;
    walkstate.level = AArch64.S2StartLevel(walkparams);
    walkstate.istable = TRUE;
    walkstate.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);

    return walkstate;
```

## Library pseudocode for aarch64/translation/vmsa\_walk/AArch64.SS2OutputPASpace

```
// AArch64.SS2OutputPASpace()
// =====
// Assign PA Space to output of Secure stage 2 translation

PASpace AArch64.SS2OutputPASpace(S2TTWParams walkparams, PASpace ipaspace)
    if ipaspace == PAS_Secure then
        if walkparams.<sw,sa> == '00' then
            return PAS_Secure;
        else
            return PAS_NonSecure;
    else
        if walkparams.<sw,sa,nsw,nsa> == '0000' then
            return PAS_Secure;
        else
            return PAS_NonSecure;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.BBMSupportLevel

```
// AArch64.BBMSupportLevel()
// =====
// Returns the level of FEAT_BBM supported

integer AArch64.BlockBBMSupportLevel()
    if !HaveBlockBBM() then
        return integer UNKNOWN;
    else
        return integer IMPLEMENTATION_DEFINED "Block BBM support level";
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.GetS1TTWParams

```
// AArch64.GetS1TTWParams()
// =====
// Returns stage 1 translation table walk parameters from respective controlling
// System registers.

S1TTWParams AArch64.GetS1TTWParams(Regime regime, SecurityState ss, bits(64) va)
    S1TTWParams walkparams;

    varange = AArch64.GetVARange(va);

    case regime of
        when Regime\_EL3    walkparams = AArch64.S1TTWParamsEL3();
        when Regime\_EL2    walkparams = AArch64.S1TTWParamsEL2(ss);
        when Regime\_EL20   walkparams = AArch64.S1TTWParamsEL20(ss, varange);
        when Regime\_EL10   walkparams = AArch64.S1TTWParamsEL10(varange);

    return walkparams;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.GetS2TTWParams

```
// AArch64.GetS2TTWParams()
// =====
// Gather walk parameters for stage 2 translation

S2TTWParams AArch64.GetS2TTWParams(SecurityState ss, PASpace ipaspace, boolean slaarch64)
    S2TTWParams walkparams;

    if ss == SS\_NonSecure then
        walkparams = AArch64.NSS2TTWParams(slaarch64);
    elsif HaveSecureEL2Ext() && ss == SS\_Secure then
        walkparams = AArch64.SS2TTWParams(ipaspace, slaarch64);
    elsif ss == SS\_Realm then
        walkparams = AArch64.RLS2TTWParams(slaarch64);
    else
        Unreachable();

    return walkparams;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.GetVARange

```
// AArch64.GetVARange()
// =====
// Determines if the VA that is to be translated lies in LOWER or UPPER address range.

VARange AArch64.GetVARange(bits(64) va)
    if va<55> == '0' then
        return VARange\_LOWER;
    else
        return VARange\_UPPER;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.HaveS1TG

```
// AArch64.HaveS1TG()
// =====
// Determine whether the given translation granule is supported for stage 1

boolean AArch64.HaveS1TG(TGx tgx)
    case tgx of
        when TGx\_4KB    return boolean IMPLEMENTATION_DEFINED "Has 4K Translation Granule";
        when TGx\_16KB   return boolean IMPLEMENTATION_DEFINED "Has 16K Translation Granule";
        when TGx\_64KB   return boolean IMPLEMENTATION_DEFINED "Has 64K Translation Granule";
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.HaveS2TG

```
// AArch64.HaveS2TG()
// =====
// Determine whether the given translation granule is supported for stage 2

boolean AArch64.HaveS2TG(TGx tgx)
    assert HaveEL(EL2);

    if HaveGTGExt() then
        case tgx of
            when TGx_4KB
                return boolean IMPLEMENTATION_DEFINED "Has Stage 2 4K Translation Granule";
            when TGx_16KB
                return boolean IMPLEMENTATION_DEFINED "Has Stage 2 16K Translation Granule";
            when TGx_64KB
                return boolean IMPLEMENTATION_DEFINED "Has Stage 2 64K Translation Granule";
        else
            return AArch64.HaveS1TG(tgx);
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.MaxTxSZ

```
// AArch64.MaxTxSZ()
// =====
// Retrieve the maximum value of TxSZ indicating minimum input address size for both
// stages of translation

integer AArch64.MaxTxSZ(TGx tgx)
    if HaveSmallTranslationTableExt() then
        case tgx of
            when TGx_4KB    return 48;
            when TGx_16KB   return 48;
            when TGx_64KB   return 47;
        else
            return 39;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.NSS2TTWParams

```
// AArch64.NSS2TTWParams()
// =====
// Gather walk parameters specific for Non-secure stage 2 translation

S2TTWParams AArch64.NSS2TTWParams(boolean slaarch64)
    S2TTWParams walkparams;

    walkparams.vm = HCR_EL2.VM OR HCR_EL2.DC;
    walkparams.tgx = AArch64.S2DecodeTG0(VTCR_EL2.TG0);
    walkparams.txsz = VTCR_EL2.T0SZ;
    walkparams.ps = VTCR_EL2.PS;
    walkparams.irgn = VTCR_EL2.IRGN0;
    walkparams.orgn = VTCR_EL2.ORGNO;
    walkparams.sh = VTCR_EL2.SH0;
    walkparams.ee = SCTLR_EL2.EE;
    walkparams.d128 = if Have128BitDescriptorExt() then VTCR_EL2.D128 else '0';
    if walkparams.d128 == '1' then
        walkparams.sk1 = VTTBR_EL2.SK1;
    else
        walkparams.sl0 = VTCR_EL2.SL0;

    walkparams.ptw = if HCR_EL2.TGE == '0' then HCR_EL2.PTW else '0';
    walkparams.fwb = if HaveStage2MemAttrControl() then HCR_EL2.FWB else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then VTCR_EL2.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then VTCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
        walkparams.ds = VTCR_EL2.DS;
    else
        walkparams.ds = '0';
    if walkparams.tgx == TGx_4KB && Have52BitIPAAndPASpaceExt() then
        walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
    else
        walkparams.sl2 = '0';
    walkparams.cmow = if HaveFeatCMOW() && IsHCRXEL2Enabled() then HCRX_EL2.CMOW else '0';
    if walkparams.d128 == '1' then
        walkparams.s2pie = '1';
    else
        walkparams.s2pie = if HaveS2PIExt() then VTCR_EL2.S2PIE else '0';
    walkparams.s2pir = if HaveS2PIExt() then S2PIR_EL2 else Zeros(64);
    if HaveTHEExt() && walkparams.d128 != '1' then
        walkparams.assuredonly = VTCR_EL2.AssuredOnly;
    else
        walkparams.assuredonly = '0';
    walkparams.tl0 = if HaveTHEExt() then VTCR_EL2.TL0 else '0';
    walkparams.tl1 = if HaveTHEExt() then VTCR_EL2.TL1 else '0';
    if HaveAccessFlagUpdateForTableExt() && walkparams.ha == '1' then
        walkparams.haft = VTCR_EL2.HAFT;
    else
        walkparams.haft = '0';

    return walkparams;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.PAMax

```
// AArch64.PAMax()
// =====
// Returns the IMPLEMENTATION_DEFINED maximum number of bits capable of representing
// physical address for this processor

integer AArch64.PAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.RLS2TTWParams

```
// AArch64.RLS2TTWParams()
// =====
// Gather walk parameters specific for Realm stage 2 translation

S2TTWParams AArch64.RLS2TTWParams(boolean slaarch64)
    // Realm stage 2 walk parameters are similar to Non-secure
    S2TTWParams walkparams = AArch64.NSS2TTWParams(slaarch64);
    walkparams.emec = if HaveFeatMEC() && IsSCTLR2EL2Enabled() then SCTLR2_EL2.EMEC else '0';
    return walkparams;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1DCacheEnabled

```
// AArch64.S1DCacheEnabled()
// =====
// Determine cacheability of stage 1 data accesses

boolean AArch64.S1DCacheEnabled(Regime regime)
    case regime of
        when Regime\_EL3 return SCTLR_EL3.C == '1';
        when Regime\_EL2 return SCTLR_EL2.C == '1';
        when Regime\_EL20 return SCTLR_EL2.C == '1';
        when Regime\_EL10 return SCTLR_EL1.C == '1';
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1DecodeTG0

```
// AArch64.S1DecodeTG0()
// =====
// Decode stage 1 granule size configuration bits TG0

TGx AArch64.S1DecodeTG0(bits(2) tg0_in)
    bits(2) tg0 = tg0_in;
    TGx tgx;

    if tg0 == '11' then
        tg0 = bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size";

    case tg0 of
        when '00' tgx = TGx\_4KB;
        when '01' tgx = TGx\_64KB;
        when '10' tgx = TGx\_16KB;

    if !AArch64.HaveS1TG(tgx) then
        case bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size" of
            when '00' tgx = TGx\_4KB;
            when '01' tgx = TGx\_64KB;
            when '10' tgx = TGx\_16KB;

    return tgx;
```



## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1DecodeTG1

```
// AArch64.S1DecodeTG1()
// =====
// Decode stage 1 granule size configuration bits TG1

TGx AArch64.S1DecodeTG1(bits(2) tgl_in)
    bits(2) tgl = tgl_in;
    TGx tgx;

    if tgl == '00' then
        tgl = bits(2) IMPLEMENTATION_DEFINED "TG1 encoded granule size";

    case tgl of
        when '10'    tgx = TGx\_4KB;
        when '11'    tgx = TGx\_64KB;
        when '01'    tgx = TGx\_16KB;

    if !AArch64.HaveS1TG(tgx) then
        case bits(2) IMPLEMENTATION_DEFINED "TG1 encoded granule size" of
            when '10'    tgx = TGx\_4KB;
            when '11'    tgx = TGx\_64KB;
            when '01'    tgx = TGx\_16KB;

    return tgx;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1E0POEnabled

```
// AArch64.S1E0POEnabled()
// =====
// Determine whether stage 1 unprivileged permission overlay is enabled

boolean AArch64.S1E0POEnabled(Regime regime, bit nv1)
    assert HasUnprivileged(regime);

    if !HaveS1POExt() then
        return FALSE;

    case regime of
        when Regime\_EL20 return IsTCR2EL2Enabled() && TCR2_EL2.E0POE == '1';
        when Regime\_EL10 return IsTCR2EL1Enabled() && nv1 == '0' && TCR2_EL1.E0POE == '1';
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1EPD

```
// AArch64.S1EPD()
// =====
// Determine whether stage 1 translation table walk is allowed for the VA range

bit AArch64.S1EPD(Regime regime, bits(64) va)
    assert HasUnprivileged(regime);
    varange = AArch64.GetVARange(va);

    case regime of
        when Regime\_EL20 return if varange == VARange\_LOWER then TCR_EL2.EPD0 else TCR_EL2.EPD1;
        when Regime\_EL10 return if varange == VARange\_LOWER then TCR_EL1.EPD0 else TCR_EL1.EPD1;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1Enabled

```
// AArch64.S1Enabled()
// =====
// Determine if stage 1 is enabled for the access type for this translation regime

boolean AArch64.S1Enabled(Regime regime, AccessType acctype)
    case regime of
        when Regime\_EL3    return SCTLR_EL3.M == '1';
        when Regime\_EL2    return SCTLR_EL2.M == '1';
        when Regime\_EL20   return SCTLR_EL2.M == '1';
        when Regime\_EL10   return (!EL2Enabled() || HCR_EL2.<DC,TGE> == '00') && SCTLR_EL1.M == '1';
```

### Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1ICacheEnabled

```
// AArch64.S1ICacheEnabled()
// =====
// Determine cacheability of stage 1 instruction fetches

boolean AArch64.S1ICacheEnabled(Regime regime)
    case regime of
        when Regime\_EL3    return SCTLRL_EL3.I == '1';
        when Regime\_EL2    return SCTLRL_EL2.I == '1';
        when Regime\_EL20   return SCTLRL_EL2.I == '1';
        when Regime\_EL10   return SCTLRL_EL1.I == '1';
```

### Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1MinTxSZ

```
// AArch64.S1MinTxSZ()
// =====
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 1

integer AArch64.S1MinTxSZ(Regime regime, bit dl28, bit ds, TGx tgx)
    if Have56BitVAExt() && dl28 == '1' then
        if HasUnprivileged(regime) then
            return 9;
        else
            return 8;
    if (Have52BitVAExt() && tgx == TGx\_64KB) || ds == '1' then
        return 12;

    return 16;
```

### Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1POEnabled

```
// AArch64.S1POEnabled()
// =====
// Determine whether stage 1 privileged permission overlay is enabled

boolean AArch64.S1POEnabled(Regime regime)
    if !HaveS1POExt() then
        return FALSE;

    case regime of
        when Regime\_EL3    return TCR_EL3.POE == '1';
        when Regime\_EL2    return IsTCR2EL2Enabled() && TCR2_EL2.POE == '1';
        when Regime\_EL20   return IsTCR2EL2Enabled() && TCR2_EL2.POE == '1';
        when Regime\_EL10   return IsTCR2EL1Enabled() && TCR2_EL1.POE == '1';
```

### Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1POR

```
// AArch64.S1POR()
// =====
// Identify stage 1 permissions overlay register for the acting translation regime

S1PORType AArch64.S1POR(Regime regime)
    case regime of
        when Regime\_EL3    return POR_EL3;
        when Regime\_EL2    return POR_EL2;
        when Regime\_EL20   return POR_EL2;
        when Regime\_EL10   return POR_EL1;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1TTBR

```
// AArch64.S1TTBR()
// =====
// Identify stage 1 table base register for the acting translation regime

bits(128) AArch64.S1TTBR(Regime regime, bits(64) va)
    varange = AArch64.GetVARange(va);

    case regime of
        when Regime\_EL3    return ZeroExtend(TTBR0_EL3, 128);
        when Regime\_EL2    return ZeroExtend(TTBR0_EL2, 128);
        when Regime\_EL20
            if varange == VARange\_LOWER then
                return ZeroExtend(TTBR0_EL2, 128);
            else
                return ZeroExtend(TTBR1_EL2, 128);
        when Regime\_EL10
            if varange == VARange\_LOWER then
                return ZeroExtend(TTBR0_EL1, 128);
            else
                return ZeroExtend(TTBR1_EL1, 128);
```



```

// AArch64.S1TTWParamsEL10()
// =====
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled)

S1TTWParams AArch64.S1TTWParamsEL10(VARange varange)
    S1TTWParams walkparams;

    if Have128BitDescriptorExt() && IsTCR2EL1Enabled() then
        walkparams.d128 = TCR2_EL1.D128;
    else
        walkparams.d128 = '0';
    if varange == VARange_LOWER then
        walkparams.tgx = AArch64.S1DecodeTG0(TCR_EL1.TG0);
        walkparams.txsz = TCR_EL1.TOSZ;
        walkparams.irgn = TCR_EL1.IRGN0;
        walkparams.orgn = TCR_EL1.ORGNO;
        walkparams.sh = TCR_EL1.SH0;
        walkparams.tbi = TCR_EL1.TBI0;

        walkparams.nfd = if HaveSVE() || HaveTME() then TCR_EL1.NFD0 else '0';
        walkparams.tbid = if HavePACEExt() then TCR_EL1.TBID0 else '0';
        walkparams.e0pd = if HaveEOPDEExt() then TCR_EL1.EOPD0 else '0';
        walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL1.HPD0 else '0';
        walkparams.mtx = if HaveMTE4Ext() then TCR_EL1.MTX0 else '0';
        walkparams.skl = if walkparams.d128 == '1' then TTBR0_EL1.SKL else '00';
        walkparams.disch = if walkparams.d128 == '1' then TCR2_EL1.DisCH0 else '0';
    else
        walkparams.tgx = AArch64.S1DecodeTG1(TCR_EL1.TG1);
        walkparams.txsz = TCR_EL1.T1SZ;
        walkparams.irgn = TCR_EL1.IRGN1;
        walkparams.orgn = TCR_EL1.ORGNO;
        walkparams.sh = TCR_EL1.SH1;
        walkparams.tbi = TCR_EL1.TBI1;

        walkparams.nfd = if HaveSVE() || HaveTME() then TCR_EL1.NFD1 else '0';
        walkparams.tbid = if HavePACEExt() then TCR_EL1.TBID1 else '0';
        walkparams.e0pd = if HaveEOPDEExt() then TCR_EL1.EOPD1 else '0';
        walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL1.HPD1 else '0';
        walkparams.mtx = if HaveMTE4Ext() then TCR_EL1.MTX1 else '0';
        walkparams.skl = if walkparams.d128 == '1' then TTBR1_EL1.SKL else '00';
        walkparams.disch = if walkparams.d128 == '1' then TCR2_EL1.DisCH1 else '0';

    walkparams.mair = MAIR_EL1;
    if HaveAIEExt() then
        walkparams.mair2 = MAIR2_EL1;
    walkparams.aie = if HaveAIEExt() && IsTCR2EL1Enabled() then TCR2_EL1.AIE else '0';
    walkparams.wxn = SCTLR_EL1.WXN;
    walkparams.ps = TCR_EL1.IPS;
    walkparams.ee = SCTLR_EL1.EE;
    if (HaveEL(EL3) && (!HaveRME() || HaveSecureEL2Ext())) then
        walkparams.sif = SCR_EL3.SIF;
    else
        walkparams.sif = '0';

    if EL2Enabled() then
        walkparams.dc = HCR_EL2.DC;
        walkparams.dct = if HaveMTE2Ext() then HCR_EL2.DCT else '0';

    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmid = SCTLR_EL1.nTlSMID;
    else
        walkparams.ntlsmid = '1';

    if EL2Enabled() then
        if HCR_EL2.<NV,NV1> == '01' then
            case ConstrainUnpredictable(Unpredictable_NVNV1) of
                when Constraint_NVNV1_00 walkparams.nv1 = '0';
                when Constraint_NVNV1_01 walkparams.nv1 = '1';
                when Constraint_NVNV1_11 walkparams.nv1 = '1';

```

```

    else
        walkparams.nv1 = HCR_EL2.NV1;
else
    walkparams.nv1 = '0';

walkparams.cmow = if HaveFeatCMOW() then SCTLR_EL1.CMOW else '0';
walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL1.HA else '0';
walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL1.HD else '0';
if walkparams.tgx IN {TGx\_4KB, TGx\_16KB} && Have52BitIPAAndPASpaceExt() then
    walkparams.ds = TCR_EL1.DS;
else
    walkparams.ds = '0';
if walkparams.d128 == '1' then
    walkparams.pie = '1';
else
    walkparams.pie = if HaveS1PIExt() && IsTCR2EL1Enabled() then TCR2_EL1.PIE else '0';
if HaveS1PIExt() then
    walkparams.pir = PIR_EL1;
    if walkparams.nv1 != '1' then
        walkparams.pire0 = PIRE0_EL1;
if HavePAN3Ext() then
    walkparams.epan = if walkparams.pie == '0' then SCTLR_EL1.EPAN else '1';
else
    walkparams.epan = '0';
if HaveTHExt() && walkparams.d128 == '0' && IsTCR2EL1Enabled() then
    walkparams.pnch = TCR2_EL1.PnCH;
else
    walkparams.pnch = '0';
if HaveAccessFlagUpdateForTableExt() && walkparams.ha == '1' && IsTCR2EL1Enabled() then
    walkparams.haft = TCR2_EL1.HAFT;
else
    walkparams.haft = '0';
walkparams.emec = if HaveFeatMEC() && IsSCTLR2EL2Enabled() then SCTLR2_EL2.EMEC else '0';

return walkparams;

```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1TTWParamsEL2

```
// AArch64.S1TTWParamsEL2()
// =====
// Gather stage 1 translation table walk parameters for EL2 regime

S1TTWParams AArch64.S1TTWParamsEL2(SecurityState ss)
    S1TTWParams walkparams;

    walkparams.tgx = AArch64.S1DecodeTG0(TCR_EL2.TG0);
    walkparams.txsz = TCR_EL2.T0SZ;
    walkparams.ps = TCR_EL2.PS;
    walkparams.irgn = TCR_EL2.IRGN0;
    walkparams.orgn = TCR_EL2.ORGNO;
    walkparams.sh = TCR_EL2.SH0;
    walkparams.tbi = TCR_EL2.TBI;
    walkparams.mair = MAIR_EL2;
    if HaveAIEExt() then
        walkparams.mair2 = MAIR2_EL2;
    walkparams.aie = if HaveAIEExt() && IsTCR2EL2Enabled() then TCR2_EL2.AIE else '0';
    walkparams.wxn = SCTLR_EL2.WXN;
    walkparams.ee = SCTLR_EL2.EE;
    if (HaveEL(EL3) && (!HaveRME() || HaveSecureEL2Ext())) then
        walkparams.sif = SCR_EL3.SIF;
    else
        walkparams.sif = '0';

    walkparams.tbid = if HavePACEExt() then TCR_EL2.TBID else '0';
    walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL2.HPD else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL2.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx\_4KB, TGx\_16KB} && Have52BitIPAAndPASpaceExt() then
        walkparams.ds = TCR_EL2.DS;
    else
        walkparams.ds = '0';
    walkparams.pie = if HaveS1PIExt() && IsTCR2EL2Enabled() then TCR2_EL2.PIE else '0';
    if HaveS1PIExt() then
        walkparams.pir = PIR_EL2;
    walkparams.mtx = if HaveMTE4Ext() then TCR_EL2.MTX else '0';
    walkparams.pnch = if HaveTHEExt() && IsTCR2EL2Enabled() then TCR2_EL2.PnCH else '0';
    if HaveAccessFlagUpdateForTableExt() && walkparams.ha == '1' && IsTCR2EL2Enabled() then
        walkparams.haft = TCR2_EL2.HAFT;
    else
        walkparams.haft = '0';
    walkparams.emec = if HaveFeatMEC() && IsSCTLR2EL2Enabled() then SCTLR2_EL2.EMEC else '0';
    if HaveFeatMEC() && ss == SS\_Realm && IsTCR2EL2Enabled() then
        walkparams.amec = TCR2_EL2.AMEC0;
    else
        walkparams.amec = '0';

    return walkparams;
```





```

// AArch64.S1TTWParamsEL20()
// =====
// Gather stage 1 translation table walk parameters for EL2&0 regime

S1TTWParams AArch64.S1TTWParamsEL20(SecurityState ss, VARange varange)
S1TTWParams walkparams;

if Have128BitDescriptorExt() && IsTCR2EL2Enabled() then
    walkparams.d128 = TCR2_EL2.D128;
else
    walkparams.d128 = '0';
if varange == VARange\_LOWER then
    walkparams.tgx = AArch64.S1DecodeTG0(TCR_EL2.TG0);
    walkparams.txsz = TCR_EL2.TOSZ;
    walkparams.irgn = TCR_EL2.IRGN0;
    walkparams.orgn = TCR_EL2.ORGNO;
    walkparams.sh = TCR_EL2.SH0;
    walkparams.tbi = TCR_EL2.TBI0;

    walkparams.nfd = if HaveSVE() || HaveTME() then TCR_EL2.NFD0 else '0';
    walkparams.tbid = if HavePACEExt() then TCR_EL2.TBID0 else '0';
    walkparams.e0pd = if HaveE0PDEExt() then TCR_EL2.E0PD0 else '0';
    walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL2.HPD0 else '0';
    walkparams.mtx = if HaveMTE4Ext() then TCR_EL2.MTX0 else '0';
    walkparams.skl = if walkparams.d128 == '1' then TTBR0_EL2.SKL else '00';
    walkparams.disch = if walkparams.d128 == '1' then TCR2_EL2.DisCH0 else '0';
else
    walkparams.tgx = AArch64.S1DecodeTG1(TCR_EL2.TG1);
    walkparams.txsz = TCR_EL2.T1SZ;
    walkparams.irgn = TCR_EL2.IRGN1;
    walkparams.orgn = TCR_EL2.ORGNO;
    walkparams.sh = TCR_EL2.SH1;
    walkparams.tbi = TCR_EL2.TBI1;

    walkparams.nfd = if HaveSVE() || HaveTME() then TCR_EL2.NFD1 else '0';
    walkparams.tbid = if HavePACEExt() then TCR_EL2.TBID1 else '0';
    walkparams.e0pd = if HaveE0PDEExt() then TCR_EL2.E0PD1 else '0';
    walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL2.HPD1 else '0';
    walkparams.mtx = if HaveMTE4Ext() then TCR_EL2.MTX1 else '0';
    walkparams.skl = if walkparams.d128 == '1' then TTBR1_EL2.SKL else '00';
    walkparams.disch = if walkparams.d128 == '1' then TCR2_EL2.DisCH1 else '0';
walkparams.mair = MAIR_EL2;
if HaveAIEExt() then
    walkparams.mair2 = MAIR2_EL2;
walkparams.aie = if HaveAIEExt() && IsTCR2EL2Enabled() then TCR2_EL2.AIE else '0';
walkparams.wxn = SCTLR_EL2.WXN;
walkparams.ps = TCR_EL2.IPS;
walkparams.ee = SCTLR_EL2.EE;
if (HaveEL(EL3) && (!HaveRME() || HaveSecureEL2Ext())) then
    walkparams.sif = SCR_EL3.SIF;
else
    walkparams.sif = '0';

if HaveTrapLoadStoreMultipleDeviceExt() then
    walkparams.ntlsmid = SCTLR_EL2.nTlSMID;
else
    walkparams.ntlsmid = '1';

walkparams.cmw = if HaveFeatCMOW() then SCTLR_EL2.CMOW else '0';
walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL2.HA else '0';
walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL2.HD else '0';
if walkparams.tgx IN {TGx\_4KB, TGx\_16KB} && Have52BitIPAAAndPASpaceExt() then
    walkparams.ds = TCR_EL2.DS;
else
    walkparams.ds = '0';
if walkparams.d128 == '1' then
    walkparams.pie = '1';
else
    walkparams.pie = if HaveS1PIExt() && IsTCR2EL2Enabled() then TCR2_EL2.PIE else '0';
if HaveS1PIExt() then

```

```

    walkparams.pir    = PIR_EL2;
    walkparams.pire0 = PIRE0_EL2;
    if HavePAN3Ext() then
        walkparams.epan = if walkparams.pie == '0' then SCTL2_EL2.EPAN else '1';
    else
        walkparams.epan = '0';
    if HaveTHExt() && walkparams.dl28 == '0' && IsTCR2EL2Enabled() then
        walkparams.pnch = TCR2_EL2.PnCH;
    else
        walkparams.pnch = '0';
    if HaveAccessFlagUpdateForTableExt() && walkparams.ha == '1' && IsTCR2EL2Enabled() then
        walkparams.haft = TCR2_EL2.HAFT;
    else
        walkparams.haft = '0';
    walkparams.emec = if HaveFeatMEC() && IsSCTL2_EL2Enabled() then SCTL2_EL2.EMEC else '0';
    if HaveFeatMEC() && ss == SS_Realm && IsTCR2EL2Enabled() then
        walkparams.amec = if varange == VARange_LOWER then TCR2_EL2.AMEC0 else TCR2_EL2.AMEC1;
    else
        walkparams.amec = '0';

    return walkparams;

```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1TTWParamsEL3

```
// AArch64.S1TTWParamsEL3()
// =====
// Gather stage 1 translation table walk parameters for EL3 regime

S1TTWParams AArch64.S1TTWParamsEL3()
    S1TTWParams walkparams;

    walkparams.tgx = AArch64.S1DecodeTG0(TCR_EL3.TG0);
    walkparams.txsz = TCR_EL3.T0SZ;
    walkparams.ps = TCR_EL3.PS;
    walkparams.irgn = TCR_EL3.IRGN0;
    walkparams.orgn = TCR_EL3.ORGNO;
    walkparams.sh = TCR_EL3.SH0;
    walkparams.tbi = TCR_EL3.TBI;
    walkparams.mair = MAIR_EL3;
    if HaveAIEExt() then
        walkparams.mair2 = MAIR2_EL3;
    walkparams.aie = if HaveAIEExt() then TCR_EL3.AIE else '0';
    walkparams.wxn = SCTLR_EL3.WXN;
    walkparams.ee = SCTLR_EL3.EE;
    walkparams.sif = if !HaveRME() || HaveSecureEL2Ext() then SCR_EL3.SIF else '0';

    walkparams.tbid = if HavePACEExt() then TCR_EL3.TBID else '0';
    walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL3.HPD else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL3.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL3.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
        walkparams.ds = TCR_EL3.DS;
    else
        walkparams.ds = '0';
    walkparams.dl28 = if Have128BitDescriptorExt() then TCR_EL3.D128 else '0';
    walkparams.skl = if walkparams.dl28 == '1' then TTBR0_EL3.SKL else '00';
    walkparams.disch = if walkparams.dl28 == '1' then TCR_EL3.DisCH0 else '0';
    if walkparams.dl28 == '1' then
        walkparams.pie = '1';
    else
        walkparams.pie = if HaveS1PIExt() then TCR_EL3.PIE else '0';
    if HaveS1PIExt() then
        walkparams.pir = PIR_EL3;
    walkparams.mtx = if HaveMTE4Ext() then TCR_EL3.MTX else '0';
    if HaveTHEExt() && walkparams.dl28 == '0' then
        walkparams.pnch = TCR_EL3.PnCH;
    else
        walkparams.pnch = '0';
    if HaveAccessFlagUpdateForTableExt() && walkparams.ha == '1' then
        walkparams.haft = TCR_EL3.HAFT;
    else
        walkparams.haft = '0';
    walkparams.emec = if HaveFeatMEC() then SCTLR2_EL3.EMEC else '0';

    return walkparams;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S2DecodeTG0

```
// AArch64.S2DecodeTG0()
// =====
// Decode stage 2 granule size configuration bits TG0

TGx AArch64.S2DecodeTG0(bits(2) tg0_in)
    bits(2) tg0 = tg0_in;
    TGx tgx;

    if tg0 == '11' then
        tg0 = bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size";

    case tg0 of
        when '00'    tgx = TGx\_4KB;
        when '01'    tgx = TGx\_64KB;
        when '10'    tgx = TGx\_16KB;

    if !AArch64.HaveS2TG(tgx) then
        case bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size" of
            when '00'    tgx = TGx\_4KB;
            when '01'    tgx = TGx\_64KB;
            when '10'    tgx = TGx\_16KB;

    return tgx;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S2MinTxSZ

```
// AArch64.S2MinTxSZ()
// =====
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 2

integer AArch64.S2MinTxSZ(bit d128, bit ds, TGx tgx, boolean slaarch64)
    ips = AArch64.PAMax();

    if d128 == '0' then
        if Have52BitPAExt() && tgx != TGx\_64KB && ds == '0' then
            ips = Min(48, AArch64.PAMax());
        else
            ips = Min(52, AArch64.PAMax());
    min_txsz = 64 - ips;
    if !slaarch64 then
        // EL1 is AArch32
        min_txsz = Min(min_txsz, 24);

    return min_txsz;
```



```

// AArch64.SS2TTWParams()
// =====
// Gather walk parameters specific for secure stage 2 translation

S2TTWParams AArch64.SS2TTWParams(PASpace ipaspace, boolean slaarch64)
    S2TTWParams walkparams;

walkparams.dl28 = if Have128BitDescriptorExt() then VTCR_EL2.Dl28 else '0';
if ipaspace == PAS_Secure then
    walkparams.tgx = AArch64.S2DecodeTG0(VTCR_EL2.TG0);
    walkparams.txs = VTCR_EL2.TOSZ;
    if walkparams.dl28 == '1' then
        walkparams.skl = VSTTBR_EL2.SKL;
    else
        walkparams.sl0 = VTCR_EL2.SL0;
    if walkparams.tgx == TGx_4KB && Have52BitIPAAndPASpaceExt() then
        walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
    else
        walkparams.sl2 = '0';
elseif ipaspace == PAS_NonSecure then
    walkparams.tgx = AArch64.S2DecodeTG0(VTCR_EL2.TG0);
    walkparams.txs = VTCR_EL2.TOSZ;
    if walkparams.dl28 == '1' then
        walkparams.skl = VTTBR_EL2.SKL;
    else
        walkparams.sl0 = VTCR_EL2.SL0;
    if walkparams.tgx == TGx_4KB && Have52BitIPAAndPASpaceExt() then
        walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
    else
        walkparams.sl2 = '0';
else
    Unreachable();

walkparams.sw = VTCR_EL2.SW;
walkparams.nsw = VTCR_EL2.NSW;
walkparams.sa = VTCR_EL2.SA;
walkparams.nsa = VTCR_EL2.NSA;
walkparams.vm = HCR_EL2.VM OR HCR_EL2.DC;
walkparams.ps = VTCR_EL2.PS;
walkparams.irgn = VTCR_EL2.IRGN0;
walkparams.orgn = VTCR_EL2.ORGNO;
walkparams.sh = VTCR_EL2.SH0;
walkparams.ee = SCTLR_EL2.EE;

walkparams.ptw = if HCR_EL2.TGE == '0' then HCR_EL2.PTW else '0';
walkparams.fwb = if HaveStage2MemAttrControl() then HCR_EL2.FWB else '0';
walkparams.ha = if HaveAccessFlagUpdateExt() then VTCR_EL2.HA else '0';
walkparams.hd = if HaveDirtyBitModifierExt() then VTCR_EL2.HD else '0';
if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
    walkparams.ds = VTCR_EL2.DS;
else
    walkparams.ds = '0';
walkparams.cmow = if HaveFeatCMOW() && IsHCRXEL2Enabled() then HCRX_EL2.CMOW else '0';
if walkparams.dl28 == '1' then
    walkparams.s2pie = '1';
else
    walkparams.s2pie = if HaveS2PIExt() then VTCR_EL2.S2PIE else '0';
walkparams.s2pir = if HaveS2PIExt() then S2PIR_EL2 else Zeros(64);
if HaveTHEExt() && walkparams.dl28 != '1' then
    walkparams.assuredonly = VTCR_EL2.AssuredOnly;
else
    walkparams.assuredonly = '0';
walkparams.tl0 = if HaveTHEExt() then VTCR_EL2.TL0 else '0';
walkparams.tl1 = if HaveTHEExt() then VTCR_EL2.TL1 else '0';
if HaveAccessFlagUpdateForTableExt() && walkparams.ha == '1' then
    walkparams.haft = VTCR_EL2.HAFT;
else
    walkparams.haft = '0';
walkparams.emec = '0';

```

```
return walkparams;
```

### Library pseudocode for aarch64/translation/vmsa\_walkparams/S2DCacheEnabled

```
// S2DCacheEnabled()
// =====
// Returns TRUE if Stage 2 Data access cacheability is enabled

boolean S2DCacheEnabled()
    return HCR_EL2.CD == '0';
```

### Library pseudocode for shared/debug/ClearStickyErrors/ClearStickyErrors

```
// ClearStickyErrors()
// =====

ClearStickyErrors()
    EDSCR.TXU = '0';           // Clear TX underrun flag
    EDSCR.RXO = '0';           // Clear RX overrun flag

    if Halted\(\) then           // in Debug state
        EDSCR.ITO = '0';         // Clear ITR overrun flag

    // If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
    // The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
    // in the pseudocode.
    if (Halted\(\) && EDSCR.ITE == '0' &&
        ConstrainUnpredictableBool\(Unpredictable\_CLEARERRITEZERO\)) then
        return;
    EDSCR.ERR = '0';           // Clear cumulative error flag

    return;
```

### Library pseudocode for shared/debug/DebugTarget/DebugTarget

```
// DebugTarget()
// =====
// Returns the debug exception target Exception level

bits(2) DebugTarget()
    ss = CurrentSecurityState\(\);
    return DebugTargetFrom(ss);
```

## Library pseudocode for shared/debug/DebugTarget/DebugTargetFrom

```
// DebugTargetFrom()
// =====

bits(2) DebugTargetFrom(SecurityState from_state)
    boolean route_to_el2;
    if HaveEL(EL2) && (from_state != SS\_Secure ||
        (HaveSecureEL2Ext() && (!HaveEL(EL3) || SCR_EL3.EEL2 == '1')) then
        if ELUsingAArch32(EL2) then
            route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
        else
            route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

    bits(2) target;
    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && !HaveAArch64() && from_state == SS\_Secure then
        target = EL3;
    else
        target = EL1;

    return target;
```

## Library pseudocode for shared/debug/DoubleLockStatus/DoubleLockStatus

```
// DoubleLockStatus()
// =====
// Returns the state of the OS Double Lock.
//     FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
//     TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()
    if !HaveDoubleLock() then
        return FALSE;
    elsif ELUsingAArch32(EL1) then
        return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
    else
        return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();
```

## Library pseudocode for shared/debug/OSLockStatus/OSLockStatus

```
// OSLockStatus()
// =====
// Returns the state of the OS Lock.

boolean OSLockStatus()
    return (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK) == '1';
```

## Library pseudocode for shared/debug/SoftwareLockStatus/Component

```
// Component
// =====
// Component Types.

enumeration Component {
    Component_PMU,
    Component_Debug,
    Component_CTI
};
```



### Library pseudocode for shared/debug/SoftwareLockStatus/GetAccessComponent

```
// GetAccessComponent()
// =====
// Returns the accessed component.

Component GetAccessComponent();
```

### Library pseudocode for shared/debug/SoftwareLockStatus/SoftwareLockStatus

```
// SoftwareLockStatus()
// =====
// Returns the state of the Software Lock.

boolean SoftwareLockStatus()
    Component component = GetAccessComponent();
    if !HaveSoftwareLock(component) then
        return FALSE;
    case component of
        when Component\_Debug
            return EDLSR.SLK == '1';
        when Component\_PMU
            return PMLSR.SLK == '1';
        when Component\_CTI
            return CTILSR.SLK == '1';
        otherwise
            Unreachable();
```

### Library pseudocode for shared/debug/authentication/AccessState

```
// AccessState()
// =====
// Returns the Security state of the access.

SecurityState AccessState();
```

## Library pseudocode for shared/debug/authentication/AllowExternalDebugAccess

```
// AllowExternalDebugAccess()
// =====
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed, FALSE otherwise.

boolean AllowExternalDebugAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    return AllowExternalDebugAccess\(AccessState\(\)\);

// AllowExternalDebugAccess()
// =====
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed for the given Security state, FALSE otherwise.

boolean AllowExternalDebugAccess(SecurityState access_state)
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveRME\(\) then
        case MDCR_EL3.<EDADE,EDAD> of
            when '00' return TRUE;
            when '01' return access_state IN {SS\_Root, SS\_Secure};
            when '10' return access_state IN {SS\_Root, SS\_Realm};
            when '11' return access_state == SS\_Root;

    if HaveSecureExtDebugView\(\) then
        if access_state == SS\_Secure then return TRUE;
    else
        if !ExternalInvasiveDebugEnabled\(\) then return FALSE;
        if ExternalSecureInvasiveDebugEnabled\(\) then return TRUE;

    if HaveEL\(EL3\) then
        EDAD_bit = if ELUsingAArch32\(EL3\) then SDCR.EDAD else MDCR_EL3.EDAD;
        return EDAD_bit == '0';
    else
        return NonSecureOnlyImplementation\(\);
```

## Library pseudocode for shared/debug/authentication/AllowExternalPMSSAccess

```
// AllowExternalPMSSAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU Snapshot
// registers is allowed, FALSE otherwise.

boolean AllowExternalPMSSAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    return AllowExternalPMSSAccess\(AccessState\(\)\);

// AllowExternalPMSSAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU Snapshot
// registers is allowed for the given Security state, FALSE otherwise.

boolean AllowExternalPMSSAccess(SecurityState access_state)
    assert IsFeatureImplemented(FEAT_PMUv3_SS) && HaveAArch64\(\);
    assert HaveSecureExtDebugView\(\); // Required when PMU Snapshot implemented

    // The access may also be subject to the OS Double Lock, power-down, etc.
    bits(2) epmssad = if HaveEL\(EL3\) then MDCR_EL3.EPMSSAD else '11';

    // Check for reserved values
    if !HaveRME\(\) && epmssad IN {'01','10'} then
        Constraint c;
        (c, epmssad) = ConstrainUnpredictableBits\(Unpredictable\_RESEPMSSAD, 2\);
        // The value returned by ConstrainUnpredictableBits() must be a
        // non-reserved value

    SecurityState allowed_state = (if HaveRME\(\) then SS\_Root else SS\_Secure);

    case epmssad of
        when '00' return access_state == allowed_state;
        when '01' return HaveRME\(\) && access_state IN {SS\_Root, SS\_Realm};
        when '10' return HaveRME\(\) && access_state IN {SS\_Root, SS\_Secure};
        when '11' return TRUE;
```

## Library pseudocode for shared/debug/authentication/AllowExternalPMUAccess

```
// AllowExternalPMUAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed, FALSE otherwise.

boolean AllowExternalPMUAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    return AllowExternalPMUAccess\(AccessState\(\)\);

// AllowExternalPMUAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed for the given Security state, FALSE otherwise.

boolean AllowExternalPMUAccess(SecurityState access_state)
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveRME\(\) then
        case MDCR_EL3.<EPMAD, EPMAD> of
            when '00' return TRUE;
            when '01' return access_state IN {SS\_Root, SS\_Secure};
            when '10' return access_state IN {SS\_Root, SS\_Realm};
            when '11' return access_state == SS\_Root;

    if HaveSecureExtDebugView\(\) then
        if access_state == SS\_Secure then return TRUE;
    else
        if !ExternalInvasiveDebugEnabled\(\) then return FALSE;
        if ExternalSecureInvasiveDebugEnabled\(\) then return TRUE;

    if HaveEL\(EL3\) then
        EPMAD_bit = if ELUsingAArch32\(EL3\) then SDCR.EPMAD else MDCR_EL3.EPMAD;
        return EPMAD_bit == '0';
    else
        return NonSecureOnlyImplementation\(\);
```

## Library pseudocode for shared/debug/authentication/AllowExternalTraceAccess

```
// AllowExternalTraceAccess()
// =====
// Returns TRUE if an external Trace access to the Trace registers is allowed, FALSE otherwise.

boolean AllowExternalTraceAccess()
    if !HaveTraceBufferExtension() then
        return TRUE;
    else
        return AllowExternalTraceAccess(AccessState());

// AllowExternalTraceAccess()
// =====
// Returns TRUE if an external Trace access to the Trace registers is allowed for the
// given Security state, FALSE otherwise.

boolean AllowExternalTraceAccess(SecurityState access_state)
    // The access may also be subject to OS lock, power-down, etc.
    if !HaveTraceBufferExtension() then return TRUE;
    assert HaveSecureExtDebugView();
    if HaveRME() then
        case MDCR_EL3.<ETADE,ETAD> of
            when '00' return TRUE;
            when '01' return access_state IN {SS_Root, SS_Secure};
            when '10' return access_state IN {SS_Root, SS_Realm};
            when '11' return access_state == SS_Root;

    if access_state == SS_Secure then return TRUE;
    if HaveEL(EL3) then
        // External Trace access is not supported for EL3 using AArch32
        assert !ELUsingAArch32(EL3);
        return MDCR_EL3.ETAD == '0';
    else
        return NonSecureOnlyImplementation();
```

## Library pseudocode for shared/debug/authentication/Debug\_authentication

```
Signal DBGEN;
Signal NIDEN;
Signal SPIDEN;
Signal SPNIDEN;
Signal RLPIDEN;
Signal RTPIDEN;
```

## Library pseudocode for shared/debug/authentication/ExternalInvasiveDebugEnabled

```
// ExternalInvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the DBGEN signal.

boolean ExternalInvasiveDebugEnabled()
    return DBGEN == Signal_High;
```

## Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugAllowed

```
// ExternalNoninvasiveDebugAllowed()
// =====
// Returns TRUE if Trace and PC Sample-based Profiling are allowed

boolean ExternalNoninvasiveDebugAllowed()
    return ExternalNoninvasiveDebugAllowed(PSTATE.EL);

// ExternalNoninvasiveDebugAllowed()
// =====

boolean ExternalNoninvasiveDebugAllowed(bits(2) el)
    if !ExternalNoninvasiveDebugEnabled() then return FALSE;
    ss = SecurityStateAtEL(el);

    if ((ELUsingAArch32(EL3) || ELUsingAArch32(EL1)) && el == EL0 &&
        ss == SS\_Secure && SDER.SUNIDEN == '1') then
        return TRUE;

    case ss of
        when SS\_NonSecure return TRUE;
        when SS\_Secure    return ExternalSecureNoninvasiveDebugEnabled();
        when SS\_Realm     return ExternalRealmNoninvasiveDebugEnabled();
        when SS\_Root      return ExternalRootNoninvasiveDebugEnabled();
```

## Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()
// =====
// This function returns TRUE if the FEAT_Debugv8p4 is implemented.
// Otherwise, this function is IMPLEMENTATION DEFINED, and, in the
// recommended interface, ExternalNoninvasiveDebugEnabled returns
// the state of the (DBGEN OR NIDEN) signal.

boolean ExternalNoninvasiveDebugEnabled()
    return !HaveNoninvasiveDebugAuth() || ExternalInvasiveDebugEnabled() || NIDEN == Signal\_High;
```

## Library pseudocode for shared/debug/authentication/ExternalRealmInvasiveDebugEnabled

```
// ExternalRealmInvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN) signal.

boolean ExternalRealmInvasiveDebugEnabled()
    if !HaveRME() then return FALSE;
    return ExternalInvasiveDebugEnabled() && RLPIDEN == Signal\_High;
```

## Library pseudocode for shared/debug/authentication/ExternalRealmNoninvasiveDebugEnabled

```
// ExternalRealmNoninvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN) signal.

boolean ExternalRealmNoninvasiveDebugEnabled()
    if !HaveRME() then return FALSE;
    return ExternalRealmInvasiveDebugEnabled();
```

## Library pseudocode for shared/debug/authentication/ExternalRootInvasiveDebugEnabled

```
// ExternalRootInvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN AND RTPIDEN AND SPIDEN) signal when FEAT_SEL2 is implemented
// and the (DBGEN AND RLPIDEN AND RTPIDEN) signal when FEAT_SEL2 is not implemented.

boolean ExternalRootInvasiveDebugEnabled()
    if !HaveRME() then return FALSE;
    return (ExternalInvasiveDebugEnabled() &&
        (!HaveSecureEL2Ext() || ExternalSecureInvasiveDebugEnabled()) &&
        ExternalRealmInvasiveDebugEnabled() &&
        RTPIDEN == Signal_High);
```

## Library pseudocode for shared/debug/authentication/ExternalRootNoninvasiveDebugEnabled

```
// ExternalRootNoninvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN AND SPIDEN AND RTPIDEN) signal.

boolean ExternalRootNoninvasiveDebugEnabled()
    if !HaveRME() then return FALSE;
    return ExternalRootInvasiveDebugEnabled();
```

## Library pseudocode for shared/debug/authentication/ExternalSecureInvasiveDebugEnabled

```
// ExternalSecureInvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN AND SPIDEN) signal.
// CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.

boolean ExternalSecureInvasiveDebugEnabled()
    if !HaveEL(EL3) && !SecureOnlyImplementation() then return FALSE;
    return ExternalInvasiveDebugEnabled() && SPIDEN == Signal_High;
```

## Library pseudocode for shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled

```
// ExternalSecureNoninvasiveDebugEnabled()
// =====
// This function returns the value of ExternalSecureInvasiveDebugEnabled() when FEAT_Debugv8p4
// is implemented. Otherwise, the definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN OR NIDEN) AND
// (SPIDEN OR SPNIDEN) signal.

boolean ExternalSecureNoninvasiveDebugEnabled()
    if !HaveEL(EL3) && !SecureOnlyImplementation() then return FALSE;
    if HaveNoninvasiveDebugAuth() then
        return (ExternalNoninvasiveDebugEnabled() &&
            (SPIDEN == Signal_High || SPNIDEN == Signal_High));
    else
        return ExternalSecureInvasiveDebugEnabled();
```

## Library pseudocode for shared/debug/authentication/IsAccessSecure

```
// IsAccessSecure()
// =====
// Returns TRUE when an access is Secure

boolean IsAccessSecure();
```

## Library pseudocode for shared/debug/authentication/IsCorePowered

```
// IsCorePowered()
// =====
// Returns TRUE if the Core power domain is powered on, FALSE otherwise.

boolean IsCorePowered();
```

## Library pseudocode for shared/debug/breakpoint/CheckValidStateMatch

```
// CheckValidStateMatch()
// =====
// Checks for an invalid state match that will generate Constrained
// Unpredictable behavior, otherwise returns Constraint_NONE.

(Constraint, bits(2), bit, bit, bits(2)) CheckValidStateMatch(bits(2) ssc_in, bit ssce_in,
                                                             bit hmc_in, bits(2) pxc_in,
                                                             boolean isbreakpnt)

    if !HaveRME() then assert ssce_in == '0';
    boolean reserved = FALSE;
    bits(2) ssc = ssc_in;
    bit ssce = ssce_in;
    bit hmc = hmc_in;
    bits(2) pxc = pxc_in;

    // Values that are not allocated in any architecture version
    case hmc:ssce:ssc:pxc of
        when '0 0 11 10' reserved = TRUE;
        when '0 0 1x xx' reserved = !HaveSecureState();
        when '1 0 00 x0' reserved = TRUE;
        when '1 0 01 10' reserved = TRUE;
        when '1 0 1x 10' reserved = TRUE;
        when 'x 1 xx xx' reserved = ssc != '01' || (hmc:pxc) IN {'000','110'};
        otherwise reserved = FALSE;

    // Match 'Usr/Sys/Svc' valid only for AArch32 breakpoints
    if (!isbreakpnt || !HaveAArch32EL(EL1)) && hmc:pxc == '000' && ssc != '11' then
        reserved = TRUE;

    // Both EL3 and EL2 are not implemented
    if !HaveEL(EL3) && !HaveEL(EL2) && (hmc != '0' || ssc != '00') then
        reserved = TRUE;

    // EL3 is not implemented
    if !HaveEL(EL3) && ssc IN {'01','10'} && hmc:ssc:pxc != '10100' then
        reserved = TRUE;

    // EL3 using AArch64 only
    if (!HaveEL(EL3) || !HaveAArch64()) && hmc:ssc:pxc == '11000' then
        reserved = TRUE;

    // EL2 is not implemented
    if !HaveEL(EL2) && hmc:ssc:pxc == '11100' then
        reserved = TRUE;

    // Secure EL2 is not implemented
    if !HaveSecureEL2Ext() && (hmc:ssc:pxc) IN {'01100','10100','x11x1'} then
        reserved = TRUE;

    if reserved then
        // If parameters are set to a reserved type, behaves as either disabled or a defined type
        Constraint c;
        (c, <hmc:ssc:ssce:pxc>) = ConstrainUnpredictableBits(Unpredictable_RESBPWPCTRL, 6);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, bits(2) UNKNOWN, bit UNKNOWN, bit UNKNOWN, bits(2) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, ssc, ssce, hmc, pxc);
```



### Library pseudocode for shared/debug/breakpoint/ContextMatchingBreakpointRange

```
// ContextMatchingBreakpointRange()
// =====
// Returns two numbers indicating the index of the first and last context-aware breakpoint.

(integer, integer) ContextMatchingBreakpointRange()
    integer b = NumBreakpointsImplemented();
    integer c = NumContextAwareBreakpointsImplemented();

    if b <= 16 then
        return (b - c, b - 1);
    elseif c <= 16 then
        return (16 - c, 15);
    else
        return (0, c - 1);
```

### Library pseudocode for shared/debug/breakpoint/IsContextMatchingBreakpoint

```
// IsContextMatchingBreakpoint()
// =====
// Returns TRUE if DBGBCR_EL1[n] is a context-aware breakpoint.

boolean IsContextMatchingBreakpoint(integer n)
    (lower, upper) = ContextMatchingBreakpointRange();
    return n >= lower && n <= upper;
```

### Library pseudocode for shared/debug/breakpoint/NumBreakpointsImplemented

```
// NumBreakpointsImplemented()
// =====
// Returns the number of breakpoints implemented.

integer NumBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of breakpoints";
```

### Library pseudocode for shared/debug/breakpoint/NumContextAwareBreakpointsImplemented

```
// NumContextAwareBreakpointsImplemented()
// =====
// Returns the number of context-aware breakpoints implemented.

integer NumContextAwareBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of context-aware breakpoints";
```

### Library pseudocode for shared/debug/breakpoint/NumWatchpointsImplemented

```
// NumWatchpointsImplemented()
// =====
// Returns the number of watchpoints implemented.

integer NumWatchpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of watchpoints";
```

### Library pseudocode for shared/debug/cti/CTI\_ProcessEvent

```
// CTI_ProcessEvent()
// =====
// Process a discrete event on a Cross Trigger output event trigger.

CTI_ProcessEvent(CrossTriggerOut id);
```

## Library pseudocode for shared/debug/cti/CTI\_SetEventLevel

```
// CTI_SetEventLevel()
// =====
// Set a Cross Trigger multi-cycle input event trigger to the specified level.

CTI_SetEventLevel(CrossTriggerIn id, Signal level);
```

## Library pseudocode for shared/debug/cti/CTI\_SignalEvent

```
// CTI_SignalEvent()
// =====
// Signal a discrete event on a Cross Trigger input event trigger.

CTI_SignalEvent(CrossTriggerIn id);
```

## Library pseudocode for shared/debug/cti/CrossTrigger

```
// CrossTrigger
// =====

enumeration CrossTriggerOut {CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest,
                             CrossTriggerOut_IRQ,           CrossTriggerOut_RSVD3,
                             CrossTriggerOut_TraceExtIn0,    CrossTriggerOut_TraceExtIn1,
                             CrossTriggerOut_TraceExtIn2,    CrossTriggerOut_TraceExtIn3};

enumeration CrossTriggerIn  {CrossTriggerIn_CrossHalt,      CrossTriggerIn_PMUOverflow,
                             CrossTriggerIn_RSVD2,          CrossTriggerIn_RSVD3,
                             CrossTriggerIn_TraceExtOut0,    CrossTriggerIn_TraceExtOut1,
                             CrossTriggerIn_TraceExtOut2,    CrossTriggerIn_TraceExtOut3};
```

## Library pseudocode for shared/debug/dccanditr/CheckForDCCInterrupts

```
// CheckForDCCInterrupts()
// =====

CheckForDCCInterrupts()
    commrx = (EDSCR.RXfull == '1');
    commtx = (EDSCR.TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    // The value to be driven onto the common COMMIRQ signal.
    boolean commirq;
    if ELUsingAArch32(EL1) then
        commirq = ((commrx && DBGDCCINT.RX == '1') ||
                   (commtx && DBGDCCINT.TX == '1'));
    else
        commirq = ((commrx && MDCCINT_EL1.RX == '1') ||
                   (commtx && MDCCINT_EL1.TX == '1'));
    SetInterruptRequestLevel(InterruptID\_COMMIRQ, if commirq then Signal\_High else Signal\_Low);

    return;
```

## Library pseudocode for shared/debug/dccanditr/DBGDTRRX\_EL0

```
// DBGDTRRX_EL0[] (external write)
// =====
// Called on writes to debug register 0x08C.

DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "generate error response";
    return;

if EDSCR.ERR == '1' then return; // Error flag set: ignore write

if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
    EDSCR.RXO = '1'; EDSCR.ERR = '1'; // Overrun condition: ignore write
    return;

EDSCR.RXfull = '1';
DTRRX = value;

if Halted() && EDSCR.MA == '1' then
    EDSCR.ITE = '0'; // See comments in EDITR[] (external write)
    if !UsingAArch32() then
        ExecuteA64(0xD5330501<31:0>); // A64 "MRS X1,DBGDTRRX_EL0"
        ExecuteA64(0xB8004401<31:0>); // A64 "STR W1,[X0],#4"
        X[1, 64] = bits(64) UNKNOWN;
    else
        ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/); // T32 "MRS R1,DBGDTRRXint"
        ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/); // T32 "STR R1,[R0],#4"
        R[1] = bits(32) UNKNOWN;
    // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
    if EDSCR.ERR == '1' then
        EDSCR.RXfull = bit UNKNOWN;
        DBGDTRRX_EL0 = bits(64) UNKNOWN;
    else
        // "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
        assert EDSCR.RXfull == '0';

        EDSCR.ITE = '1'; // See comments in EDITR[] (external write)
    return;

// DBGDTRRX_EL0[] (external read)
// =====

bits(32) DBGDTRRX_EL0[boolean memory_mapped]
return DTRRX;
```

## Library pseudocode for shared/debug/dccanditr/DBGDTRTX\_EL0

```
// DBGDTRTX_EL0[] (external read)
// =====
// Called on reads of debug register 0x080.

bits(32) DBGDTRTX_EL0[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                                // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return bits(32) UNKNOWN;

    underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
    value = if underrun then bits(32) UNKNOWN else DTRTX;

    if EDSCR.ERR == '1' then return value;                        // Error flag set: no side-effects

    if underrun then
        EDSCR.TXU = '1'; EDSCR.ERR = '1';                        // Underrun condition: block side-effects
        return value;                                            // Return UNKNOWN

    EDSCR.TXfull = '0';
    if Halted() && EDSCR.MA == '1' then                            // See comments in EDITR[] (external write)
        EDSCR.ITE = '0';

    if !UsingAArch32() then
        ExecuteA64(0xB8404401<31:0>);                            // A64 "LDR W1,[X0],#4"
    else
        ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/); // T32 "LDR R1,[R0],#4"
    // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
    if EDSCR.ERR == '1' then
        EDSCR.TXfull = bit UNKNOWN;
        DBGDTRTX_EL0 = bits(64) UNKNOWN;
    else
        if !UsingAArch32() then
            ExecuteA64(0xD5130501<31:0>);                        // A64 "MSR DBGDTRTX_EL0,X1"
        else
            ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/); // T32 "MSR DBGDTRTXint,R1"
            // "MSR DBGDTRTX_EL0,X1" calls DBGDTR_EL0[] (write) which sets TXfull.
            assert EDSCR.TXfull == '1';
        if !UsingAArch32() then
            X[1, 64] = bits(64) UNKNOWN;
        else
            R[1] = bits(32) UNKNOWN;
        EDSCR.ITE = '1';                                          // See comments in EDITR[] (external write)

    return value;

// DBGDTRTX_EL0[] (external write)
// =====

DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value
    DTRTX = value;
    return;
```

## Library pseudocode for shared/debug/dccanditr/DBGDTR\_EL0

```
// DBGDTR_EL0[] (write)
// =====
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

DBGDTR_EL0[] = bits(N) value_in
  bits(N) value = value_in;
  // For MSR DBGDTRTX_EL0,<Rt>  N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
  // For MSR DBGDTR_EL0,<Xt>    N=64, value=X[t]<63:0>
  assert N IN {32,64};
  if EDSCR.TXfull == '1' then
    value = bits(N) UNKNOWN;
  // On a 64-bit write, implement a half-duplex channel
  if N == 64 then DTRRX = value<63:32>;
  DTRTX = value<31:0>;          // 32-bit or 64-bit write
  EDSCR.TXfull = '1';
  return;

// DBGDTR_EL0[] (read)
// =====
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) DBGDTR_EL0[]
  // For MRS <Rt>,DBGDTRTX_EL0  N=32, X[t]=Zeros(32):result
  // For MRS <Xt>,DBGDTR_EL0    N=64, X[t]=result
  assert N IN {32,64};
  bits(N) result;
  if EDSCR.RXfull == '0' then
    result = bits(N) UNKNOWN;
  else
    // On a 64-bit read, implement a half-duplex channel
    // NOTE: the word order is reversed on reads with regards to writes
    if N == 64 then result<63:32> = DTRTX;
    result<31:0> = DTRRX;
  EDSCR.RXfull = '0';
  return result;
```

## Library pseudocode for shared/debug/dccanditr/DTR

```
bits(32) DTRRX;
bits(32) DTRTX;
```

## Library pseudocode for shared/debug/dccanditr/EDITR

```
// EDITR[] (external write)
// =====
// Called on writes to debug register 0x084.

EDITR[boolean memory_mapped] = bits(32) value
  if EDPRSR<6:5,0> != '001' then                                // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "generate error response";
    return;

  if EDSCR.ERR == '1' then return;                                // Error flag set: ignore write

  if !Halted() then return;                                       // Non-debug state: ignore write

  if EDSCR.ITE == '0' || EDSCR.MA == '1' then
    EDSCR.ITO = '1'; EDSCR.ERR = '1';                             // Overrun condition: block write
    return;

  // ITE indicates whether the processor is ready to accept another instruction; the processor
  // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
  // is no indication that the pipeline is empty (all instructions have completed). In this
  // pseudocode, the assumption is that only one instruction can be executed at a time,
  // meaning ITE acts like "InstrCompl".
  EDSCR.ITE = '0';

  if !UsingAArch32() then
    ExecuteA64(value);
  else
    ExecuteT32(value<15:0> /*hw1*/, value<31:16> /*hw2*/);

  EDSCR.ITE = '1';

return;
```



```

// DCPSInstruction()
// =====
// Operation of the DCPS instruction in Debug state

DCPSInstruction(bits(2) target_el)

    SynchronizeContext();

bits(2) handle_el;
case target_el of
    when EL1
        if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then
            handle_el = PSTATE.EL;
        elsif EL2Enabled() && HCR_EL2.TGE == '1' then
            UNDEFINED;
        else
            handle_el = EL1;
    when EL2
        if !HaveEL(EL2) then
            UNDEFINED;
        elsif PSTATE.EL == EL3 && !UsingAArch32() then
            handle_el = EL3;
        elsif !IsSecureEL2Enabled() && CurrentSecurityState() == SS_Secure then
            UNDEFINED;
        else
            handle_el = EL2;
    when EL3
        if EDSCR.SDD == '1' || !HaveEL(EL3) then
            UNDEFINED;
        else
            handle_el = EL3;
    otherwise
        Unreachable();

from_secure = CurrentSecurityState() == SS_Secure;
if ELUsingAArch32(handle_el) then
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    assert UsingAArch32(); // Cannot move from AArch64 to AArch32
    case handle_el of
        when EL1
            AArch32.WriteMode(M32_Svc);
            if HavePANExt() && SCTL.R.SPAN == '0' then
                PSTATE.PAN = '1';
        when EL2 AArch32.WriteMode(M32_Hyp);
        when EL3
            AArch32.WriteMode(M32_Monitor);
            if HavePANExt() then
                if !from_secure then
                    PSTATE.PAN = '0';
                elsif SCTL.R.SPAN == '0' then
                    PSTATE.PAN = '1';
    if handle_el == EL2 then
        ELR_hyp = bits(32) UNKNOWN; HSR = bits(32) UNKNOWN;
    else
        LR = bits(32) UNKNOWN;
        SPSR_curr[] = bits(32) UNKNOWN;
        PSTATE.E = SCTL.R.ELx[].EE;
        DLR = bits(32) UNKNOWN; DSPSR = bits(32) UNKNOWN;

else // Targeting AArch64
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();
    if from_32 && HaveSME() && PSTATE.SM == '1' then
        ResetSVEState();
    else
        MaybeZeroSVEUppers(target_el);
    PSTATE.nRW = '0'; PSTATE.SP = '1'; PSTATE.EL = handle_el;
    if HavePANExt() && ((handle_el == EL1 && SCTL.R_EL1.SPAN == '0') ||
        (handle_el == EL2 && HCR_EL2.E2H == '1' &&
        HCR_EL2.TGE == '1' && SCTL.R_EL2.SPAN == '0')) then

```



```

    PSTATE.PAN = '1';
    ELR_ELx[] = bits(64) UNKNOWN; SPSR_ELx[] = bits(64) UNKNOWN; ESR_ELx[] = bits(64) UNKNOWN;
    DLR_ELO = bits(64) UNKNOWN; DSPSR_ELO = bits(64) UNKNOWN;
    if HaveUAOExt() then PSTATE.UAO = '0';
    if HaveMTEExt() then PSTATE.TCO = '1';
    if HaveGCS() then PSTATE.EXLOCK = '0';

    UpdateEDSCRFields(); // Update EDSCR PE state flags
    sync_errors = HaveIESB() && SCTLRL_ELx[].IESB == '1';
    if HaveDoubleFaultExt() && !UsingAArch32() then
        sync_errors = (sync_errors ||
            (EffectiveEA() == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3));
    // SCTLRL_ELx[].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();
    return;

```

### Library pseudocode for shared/debug/halting/DRPSInstruction

```

// DRPSInstruction()
// =====
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()

    sync_errors = HaveIESB() && SCTLRL_ELx[].IESB == '1';
    if HaveDoubleFaultExt() && !UsingAArch32() then
        sync_errors = (sync_errors ||
            (EffectiveEA() == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3));
    // SCTLRL_ELx[].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();

    SynchronizeContext();

    DebugRestorePSR();

    return;

```

### Library pseudocode for shared/debug/halting/DebugHalt

```

constant bits(6) DebugHalt_Breakpoint      = '000111';
constant bits(6) DebugHalt_EDBGRQ         = '010011';
constant bits(6) DebugHalt_Step_Normal     = '011011';
constant bits(6) DebugHalt_Step_Exclusive = '011111';
constant bits(6) DebugHalt_OSUnlockCatch   = '100011';
constant bits(6) DebugHalt_ResetCatch      = '100111';
constant bits(6) DebugHalt_Watchpoint      = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess  = '110011';
constant bits(6) DebugHalt_ExceptionCatch  = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';

```

## Library pseudocode for shared/debug/halting/DebugRestorePSR

```
// DebugRestorePSR()
// =====

DebugRestorePSR()
    // PSTATE.{N,Z,C,V,Q,GE,SS,D,A,I,F} are not observable and ignored in Debug state, so
    // behave as if UNKNOWN.
    if UsingAArch32() then
        bits(32) spsr = SPSR_curr[];
        SetPSTATEFromPSR(spsr);
        PSTATE.<N,Z,C,V,Q,GE,SS,A,I,F> = bits(13) UNKNOWN;
        // In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000'; PSTATE.T = '1'; // PSTATE.J is RES0
        DLR = bits(32) UNKNOWN; DSPSR = bits(32) UNKNOWN;
    else
        bits(64) spsr = SPSR_ELx[];
        SetPSTATEFromPSR(spsr);
        PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN; DSPSR_EL0 = bits(64) UNKNOWN;
        UpdateEDSCRFields(); // Update EDSCR PE state flags
```

## Library pseudocode for shared/debug/halting/DisableITRAndResumeInstructionPrefetch

```
// DisableITRAndResumeInstructionPrefetch()
// =====

DisableITRAndResumeInstructionPrefetch();
```

## Library pseudocode for shared/debug/halting/ExecuteA64

```
// ExecuteA64()
// =====
// Execute an A64 instruction in Debug state.

ExecuteA64(bits(32) instr);
```

## Library pseudocode for shared/debug/halting/ExecuteT32

```
// ExecuteT32()
// =====
// Execute a T32 instruction in Debug state.

ExecuteT32(bits(16) hw1, bits(16) hw2);
```

## Library pseudocode for shared/debug/halting/ExitDebugState

```
// ExitDebugState()
// =====

ExitDebugState()
    assert Halted();
    SynchronizeContext();

    // Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
    // detect that the PE has restarted.
    EDSCR.STATUS = '000001'; // Signal restarting
    // Clear any pending Halting debug events
    if Havev8p8Debug() then
        EDESR<3:0> = '0000';
    else
        EDESR<2:0> = '000';

    bits(64) new_pc;
    bits(64) spsr;

    if UsingAArch32() then
        new_pc = ZeroExtend(DLR, 64);
        if Havev8p9Debug() then
            spsr = DSPSR2 : DSPSR;
        else
            spsr = ZeroExtend(DSPSR, 64);
    else
        new_pc = DLR_EL0;
        spsr = DSPSR_EL0;

    boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    SetPSTATEFromPSR(spsr); // Can update privileged bits, even at EL0

    boolean branch_conditional = FALSE;
    if UsingAArch32() then
        if ConstrainUnpredictableBool(Unpredictable\_RESTARTALIGNPC) then new_pc<0> = '0';
        // AArch32 branch
        BranchTo(new_pc<31:0>, BranchType\_DBGEXIT, branch_conditional);
    else
        // If targeting AArch32 then PC[63:32,1:0] might be set to UNKNOWN.
        if illegal_psr_state && spsr<4> == '1' then
            new_pc<63:32> = bits(32) UNKNOWN;
            new_pc<1:0> = bits(2) UNKNOWN;
        if HaveBRBExt() then
            BRBEDebugStateExit(new_pc);
        // A type of branch that is never predicted
        BranchTo(new_pc, BranchType\_DBGEXIT, branch_conditional);

    (EDSCR.STATUS, EDPRSR.SDR) = ('000010', '1'); // Atomically signal restarted
    EDPRSR.HALTED = '0';
    UpdateEDSCRFields(); // Stop signalling PE state
    DisableITRAndResumeInstructionPrefetch();

    return;
```



```

// Halt()
// =====

Halt(bits(6) reason)
    boolean is_async = FALSE;
    FaultRecord fault = NoFault();
    Halt(reason, is_async, fault);

// Halt()
// =====

Halt(bits(6) reason, boolean is_async, FaultRecord fault)
    if HaveTME() && TSTATE.depth > 0 then
        FailTransaction(TMFailure\_DBG, FALSE);

    CTI\_SignalEvent(CrossTriggerIn\_CrossHalt); // Trigger other cores to halt

    bits(64) preferred_restart_address = ThisInstrAddr(64);
    bits(64) spsr = GetPSRFromPSTATE(DebugState, 64);

    if (HaveBTIEExt() && !is_async && !(reason IN {DebugHalt\_Step\_Normal, DebugHalt\_Step\_Exclusive,
DebugHalt\_Step\_NoSyndrome, DebugHalt\_Breakpoint, DebugHalt\_HaltInstruction})) &&
ConstrainUnpredictableBool(Unpredictable\_ZEROBTYP)) then
        spsr<11:10> = '00';

    if UsingAArch32() then
        DLR = preferred_restart_address<31:0>;
        DSPSR = spsr<31:0>;
        if Havev8p9Debug() then
            DSPSR2 = spsr<63:32>;
    else
        DLR_EL0 = preferred_restart_address;
        DSPSR_EL0 = spsr;

    EDSCR.ITE = '1';
    EDSCR.ITO = '0';
    if HaveRME() then
        if PSTATE.EL == EL3 then
            EDSCR.SDD = '0';
        else
            EDSCR.SDD = if ExternalRootInvasiveDebugEnabled() then '0' else '1';
    elsif CurrentSecurityState() == SS\_Secure then
        EDSCR.SDD = '0'; // If entered in Secure state, allow debug
    elsif HaveEL(EL3) then
        EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
    else
        EDSCR.SDD = '1'; // Otherwise EDSCR.SDD is RES1
    EDSCR.MA = '0';

    // In Debug state:
    // * PSTATE.{SS,SSBS,D,A,I,F} are not observable and ignored so behave-as-if UNKNOWN.
    // * PSTATE.{N,Z,C,V,Q,GE,E,M,nRW,EL,SP,DIT} are also not observable, but since these
    //     are not changed on exception entry, this function also leaves them unchanged.
    // * PSTATE.{IT,T} are ignored.
    // * PSTATE.IL is ignored and behave-as-if 0.
    // * PSTATE.BTYPE is ignored and behave-as-if 0.
    // * PSTATE.TCO is set 1.
    // * PSTATE.{UAO,PAN} are observable and not changed on entry into Debug state.

    if UsingAArch32() then
        PSTATE.<IT,SS,SSBS,A,I,F,T> = bits(14) UNKNOWN;
    else
        PSTATE.<SS,SSBS,D,A,I,F> = bits(6) UNKNOWN;

        PSTATE.TCO = '1';
        PSTATE.BTYPE = '00';
    PSTATE.IL = '0';
    StopInstructionPrefetchAndEnableITR();
    (EDSCR.STATUS,EDPSR.HALTED) = (reason,'1');
    UpdateEDSCRFields(); // Update EDSCR PE state flags.

```

```

if IsFeatureImplemented(FEAT_EDHSR) then
    UpdateEDHSR(reason, fault);           // Update EDHSR fields.
if !is_async then EndOfInstruction();
return;

```

### Library pseudocode for shared/debug/halting/HaltOnBreakpointOrWatchpoint

```

// HaltOnBreakpointOrWatchpoint()
// =====
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';

```

### Library pseudocode for shared/debug/halting/Halted

```

// Halted()
// =====

boolean Halted()
    return !(EDSCR.STATUS IN {'000001', '000010'});           // Halted

```

### Library pseudocode for shared/debug/halting/HaltingAllowed

```

// HaltingAllowed()
// =====
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    ss = CurrentSecurityState();
    case ss of
        when SS\_NonSecure return ExternalInvasiveDebugEnabled();
        when SS\_Secure    return ExternalSecureInvasiveDebugEnabled();
        when SS\_Root     return ExternalRootInvasiveDebugEnabled();
        when SS\_Realm    return ExternalRealmInvasiveDebugEnabled();

```

### Library pseudocode for shared/debug/halting/Restarting

```

// Restarting()
// =====

boolean Restarting()
    return EDSCR.STATUS == '000001';           // Restarting

```

### Library pseudocode for shared/debug/halting/StopInstructionPrefetchAndEnableITR

```

// StopInstructionPrefetchAndEnableITR()
// =====

StopInstructionPrefetchAndEnableITR();

```

## Library pseudocode for shared/debug/halting/UpdateDbgAuthStatus

```
// UpdateDbgAuthStatus()
// =====
// Provides information about the state of the
// implementation defined authentication interface for debug.

UpdateDbgAuthStatus()
    bits(2) nsid, nsnid;
    bits(2) sid, snid;
    bits(32) regVal = Zeros(32);
    if HaveEL(EL3) then
        if ExternalInvasiveDebugEnabled() then
            nsid = '11'; // Non-Secure Invasive debug implemented and enabled.
        else
            nsid = '10'; // Non-Secure Invasive debug implemented and disabled.
        if IsFeatureImplemented(FEAT_Debugv8p4) || ExternalNoninvasiveDebugEnabled() then
            nsnid = '11'; // Non-Secure Non-Invasive debug implemented and enabled.
        else
            nsnid = '10'; // Non-Secure Non-Invasive debug implemented and disabled.
        if ExternalSecureInvasiveDebugEnabled() then
            sid = '11'; // Secure Invasive debug implemented and enabled.
        else
            sid = '10'; // Secure Invasive debug implemented and disabled.
        if IsFeatureImplemented(FEAT_Debugv8p4) || ExternalSecureNoninvasiveDebugEnabled() then
            snid = '11'; // Field has the same value as DBGAUTHSTATUS_EL1.SID
        else
            snid = '10'; // Secure Non-Invasive debug implemented and disabled.
    else
        sid = '00';
        snid = '00';
        nsid = '00';
        nsnid = '00';

    DBGAUTHSTATUS_EL1.NSID = nsid;
    DBGAUTHSTATUS_EL1.NSNID = nsnid;
    DBGAUTHSTATUS_EL1.SID = sid;
    DBGAUTHSTATUS_EL1.SNID = snid;
    return;
```

## Library pseudocode for shared/debug/halting/UpdateEDHSR

```
// UpdateEDHSR()
// =====
// Update EDHSR watchpoint related fields.

UpdateEDHSR(bits(6) reason, FaultRecord fault)
    bits(64) syndrome = Zeros(64);
    if reason == DebugHalt\_Watchpoint then
        if HaveGCS() && fault.accessdesc.acctype == AccessType\_GCS then
            syndrome<40> = '1'; // GCS
            syndrome<23:0> = WatchpointRelatedSyndrome(fault, EDWAR);
        else
            syndrome = bits(64) UNKNOWN;
    EDHSR = syndrome;
```

## Library pseudocode for shared/debug/halting/UpdateEDSCRFields

```
// UpdateEDSCRFields()
// =====
// Update EDSCR PE state fields

UpdateEDSCRFields()

if !Halted() then
    EDSCR.EL = '00';
    if HaveRME() then
        // SDD bit.
        EDSCR.SDD = if ExternalRootInvasiveDebugEnabled() then '0' else '1';
        EDSCR.<NSE,NS> = bits(2) UNKNOWN;
    else
        // SDD bit.
        EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
        EDSCR.NS = bit UNKNOWN;

    EDSCR.RW = '1111';
else
    EDSCR.EL = PSTATE.EL;
    // Error Pending.
    if EL2Enabled() && HCR_EL2.<AMO,TGE> == '10' && PSTATE.EL IN {EL0,EL1} then
        EDSCR.A = if IsVirtualSErrorPending() then '1' else '0';
    else
        EDSCR.A = if IsPhysicalSErrorPending() then '1' else '0';

    ss = CurrentSecurityState();
    if HaveRME() then
        case ss of
            when SS_Secure      EDSCR.<NSE,NS> = '00';
            when SS_NonSecure    EDSCR.<NSE,NS> = '01';
            when SS_Root         EDSCR.<NSE,NS> = '10';
            when SS_Realm        EDSCR.<NSE,NS> = '11';
    else
        EDSCR.NS = if ss == SS_Secure then '0' else '1';

    bits(4) RW;
    RW<1> = if ELUsingAArch32(EL1) then '0' else '1';
    if PSTATE.EL != EL0 then
        RW<0> = RW<1>;
    else
        RW<0> = if UsingAArch32() then '0' else '1';
    if !HaveEL(EL2) || (HaveEL(EL3) && SCR_curr[].NS == '0' && !IsSecureEL2Enabled()) then
        RW<2> = RW<1>;
    else
        RW<2> = if ELUsingAArch32(EL2) then '0' else '1';
    if !HaveEL(EL3) then
        RW<3> = RW<2>;
    else
        RW<3> = if ELUsingAArch32(EL3) then '0' else '1';

    // The least-significant bits of EDSCR.RW are UNKNOWN if any higher EL is using AArch32.
    if RW<3> == '0' then RW<2:0> = bits(3) UNKNOWN;
    elsif RW<2> == '0' then RW<1:0> = bits(2) UNKNOWN;
    elsif RW<1> == '0' then RW<0> = bit UNKNOWN;
    EDSCR.RW = RW;
return;
```



## Library pseudocode for shared/debug/haltingevents/CheckExceptionCatch

```
// CheckExceptionCatch()
// =====
// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch(boolean exception_entry)
    // Called after an exception entry or exit, that is, such that the Security state
    // and PSTATE.EL are correct for the exception target. When FEAT_Debugv8p2
    // is not implemented, this function might also be called at any time.
    ss = SecurityStateAtEL(PSTATE.EL);
    integer base;

    case ss of
        when SS\_Secure      base = 0;
        when SS\_NonSecure  base = 4;
        when SS\_Realm      base = 16;
        when SS\_Root       base = 0;
    if HaltingAllowed() then
        boolean halt;
        if HaveExtendedECDebugEvents() then
            exception_exit = !exception_entry;
            increment = if ss == SS\_Realm then 4 else 8;
            ctrl = EDECCR<UInt(PSTATE.EL) + base + increment>:EDECCR<UInt(PSTATE.EL) + base>;
            case ctrl of
                when '00'  halt = FALSE;
                when '01'  halt = TRUE;
                when '10'  halt = (exception_exit == TRUE);
                when '11'  halt = (exception_entry == TRUE);
            else
                halt = (EDECCR<UInt(PSTATE.EL) + base> == '1');

            if halt then
                if Havev8p8Debug() && exception_entry then
                    EDESR.EC = '1';
                else
                    Halt(DebugHalt\_ExceptionCatch);
```

## Library pseudocode for shared/debug/haltingevents/CheckHaltingStep

```
// CheckHaltingStep()
// =====
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep(boolean is_async)
    step_enabled = EDECR.SS == '1' && HaltingAllowed();
    active_pending = step_enabled && EDESR.SS == '1';
    if active_pending then
        if HaltingStep\_DidNotStep() then
            FaultRecord fault = NoFault();
            Halt(DebugHalt\_Step\_NoSyndrome, is_async, fault);
        elsif HaltingStep\_SteppedEX() then
            FaultRecord fault = NoFault();
            Halt(DebugHalt\_Step\_Exclusive, is_async, fault);
        else
            FaultRecord fault = NoFault();
            Halt(DebugHalt\_Step\_Normal, is_async, fault);
    if step_enabled then ShouldAdvanceHS = TRUE;
    return;
```

### Library pseudocode for shared/debug/haltingevents/CheckOSUnlockCatch

```
// CheckOSUnlockCatch()
// =====
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()
    if ((HaveDoPD() && CTIDEVCTL.OSUCE == '1') ||
        (!HaveDoPD() && EDECR.OSUCE == '1')) then
        if !Halted() then EDESR.OSUC = '1';
```

### Library pseudocode for shared/debug/haltingevents/CheckPendingExceptionCatch

```
// CheckPendingExceptionCatch()
// =====
// Check whether EDESR.EC has been set by an Exception Catch debug event.

CheckPendingExceptionCatch(boolean is_async)
    if Havev8p8Debug() && HaltingAllowed() && EDESR.EC == '1' then
        FaultRecord fault = NoFault();
        Halt(DebugHalt_ExceptionCatch, is_async, fault);
```

### Library pseudocode for shared/debug/haltingevents/CheckPendingOSUnlockCatch

```
// CheckPendingOSUnlockCatch()
// =====
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
    if HaltingAllowed() && EDESR.OSUC == '1' then
        boolean is_async = TRUE;
        FaultRecord fault = NoFault();
        Halt(DebugHalt_OSUnlockCatch, is_async, fault);
```

### Library pseudocode for shared/debug/haltingevents/CheckPendingResetCatch

```
// CheckPendingResetCatch()
// =====
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        boolean is_async = TRUE;
        FaultRecord fault = NoFault();
        Halt(DebugHalt_ResetCatch, is_async, fault);
```

### Library pseudocode for shared/debug/haltingevents/CheckResetCatch

```
// CheckResetCatch()
// =====
// Called after reset

CheckResetCatch()
    if (HaveDoPD() && CTIDEVCTL.RCE == '1') || (!HaveDoPD() && EDECR.RCE == '1') then
        EDESR.RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch);
```

## Library pseudocode for shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters

```
// CheckSoftwareAccessToDebugRegisters()
// =====
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
    os_lock = (if ELUsingAArch32\(EL1\) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK);
    if HaltingAllowed\(\) && EDSCR.TDA == '1' && os_lock == '0' then
        Halt\(DebugHalt\_SoftwareAccess\);
```

## Library pseudocode for shared/debug/haltingevents/CheckTRBEHalt

```
// CheckTRBEHalt()
// =====

CheckTRBEHalt()
    if !Havev8p9Debug\(\) || !HaveFeatTRBEExt\(\) then
        return;

    if (HaltingAllowed\(\) && TraceBufferEnabled\(\) &&
        TRBSR_EL1.IRQ == '1' && EDECR.TRBE == '1') then
        Halt\(DebugHalt\_EDBGRQ\);
```

## Library pseudocode for shared/debug/haltingevents/ExternalDebugRequest

```
// ExternalDebugRequest()
// =====

ExternalDebugRequest()
    if HaltingAllowed\(\) then
        boolean is_async = TRUE;
        FaultRecord fault = NoFault\(\);
        Halt\(DebugHalt\_EDBGRQ, is\_async, fault\);
    // Otherwise the CTI continues to assert the debug request until it is taken.
```

## Library pseudocode for shared/debug/haltingevents/HSAdvance

```
// HSAdvance()
// =====
// Advance the Halting Step State Machine

HSAdvance()
    if !ShouldAdvanceHS then return;
    step_enabled = EDECR.SS == '1' && HaltingAllowed\(\);
    active_not_pending = step_enabled && EDESR.SS == '0';
    if active_not_pending then EDESR.SS = '1'; // set as pending.
    ShouldAdvanceHS = FALSE;
    return;
```

## Library pseudocode for shared/debug/haltingevents/HaltingStep\_DidNotStep

```
// HaltingStep_DidNotStep()
// =====
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.

boolean HaltingStep_DidNotStep();
```

## Library pseudocode for shared/debug/haltingevents/HaltingStep\_SteppedEX

```
// HaltingStep_SteppedEX()
// =====
// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.

boolean HaltingStep_SteppedEX();
```

## Library pseudocode for shared/debug/interrupts/ExternalDebugInterruptsDisabled

```
// ExternalDebugInterruptsDisabled()
// =====
// Determine whether EDSCR disables interrupts routed to 'target'.

boolean ExternalDebugInterruptsDisabled(bits(2) target)
    boolean int_dis;
    SecurityState ss = SecurityStateAtEL(target);
    if Havev8p4Debug() then
        if EDSCR.INTdis[0] == '1' then
            case ss of
                when SS\_NonSecure int_dis = ExternalInvasiveDebugEnabled();
                when SS\_Secure int_dis = ExternalSecureInvasiveDebugEnabled();
                when SS\_Realm int_dis = ExternalRealmInvasiveDebugEnabled();
                when SS\_Root int_dis = ExternalRootInvasiveDebugEnabled();
            else
                int_dis = FALSE;
        else
            case target of
                when EL3
                    int_dis = (EDSCR.INTdis == '11' && ExternalSecureInvasiveDebugEnabled());
                when EL2
                    int_dis = (EDSCR.INTdis IN {'1x'} && ExternalInvasiveDebugEnabled());
                when EL1
                    if ss == SS\_Secure then
                        int_dis = (EDSCR.INTdis IN {'1x'} && ExternalSecureInvasiveDebugEnabled());
                    else
                        int_dis = (EDSCR.INTdis != '00' && ExternalInvasiveDebugEnabled());
            return int_dis;
```

## Library pseudocode for shared/debug/pmu

```
array integer PMUEventAccumulator[0..30]; // Accumulates PMU events for a cycle

array boolean PMULastThresholdValue[0..30]; // A record of the threshold result for each
```

## Library pseudocode for shared/debug/pmu/CYCLE\_COUNTER\_ID

```
constant integer CYCLE_COUNTER_ID = 31;
```

## Library pseudocode for shared/debug/pmu/CheckForPMUOverflow

```
// CheckForPMUOverflow()
// =====
// Signal Performance Monitors overflow IRQ and CTI overflow events.
// Called before each instruction is executed.

CheckForPMUOverflow()
    boolean check_cnten = FALSE;
    boolean check_e      = TRUE;
    boolean check_inten  = TRUE;
    boolean include_lo   = TRUE;
    boolean include_hi   = TRUE;
    boolean exclude_cyc  = FALSE;
    boolean exclude_sync = FALSE;

    boolean enabled = PMUIRQEnabled\(\);
    boolean pmuirq = PMUOverflowCondition(check_e, check_cnten, check_inten,
                                         include_hi, include_lo,
                                         exclude_cyc, exclude_sync);

    SetInterruptRequestLevel(InterruptID\_PMIIRQ,
                             if enabled && pmuirq then Signal\_High else Signal\_Low);
    CTI\_SetEventLevel(CrossTriggerIn\_PMUOverflow, if pmuirq then Signal\_High else Signal\_Low);

    // The request remains set until the condition is cleared.
    // For example, an interrupt handler or cross-triggered event handler clears
    // the overflow status flag by writing to PMOVSLR_EL0.

    if HavePMUv3p9\(\) && Havev8p9Debug() then
        if pmuirq && HaltingAllowed() && EDECR.PME == '1' then
            Halt(DebugHalt\_EDBGRQ);

    if ShouldBRBEFreeze() then
        BRBEFreeze();

    return;
```



```

// CountPMUEvents()
// =====
// Return TRUE if counter "idx" should count its event.
// For the cycle counter, idx == CYCLE_COUNTER_ID (32).
// For the instruction counter, idx == INSTRUCTION_COUNTER_ID (33).

boolean CountPMUEvents(integer idx)
    constant integer num_counters = GetNumEventCounters\(\);
    assert (idx == CYCLE\_COUNTER\_ID || idx < num_counters ||
            (idx == INSTRUCTION\_COUNTER\_ID && HavePMUv3ICNTR\(\)));

    boolean debug;
    boolean enabled;
    boolean prohibited;
    boolean filtered;
    boolean frozen;
    boolean resvd_for_el2;
    bit E;

    // Event counting is disabled in Debug state
    debug = Halted\(\);

    // Software can reserve some counters for EL2
    resvd_for_el2 = PMUCounterIsHyp(idx);
    ss = CurrentSecurityState();

    // Main enable controls
    case idx of
        when INSTRUCTION\_COUNTER\_ID
            assert HaveAArch64();
            enabled = PMCR_EL0.E == '1' && PMCNTENSET_EL0.F0 == '1';
        when CYCLE\_COUNTER\_ID
            if HaveAArch64() then
                enabled = PMCR_EL0.E == '1' && PMCNTENSET_EL0.C == '1';
            else
                enabled = PMCR.E == '1' && PMCNTENSET.C == '1';
        otherwise
            if resvd_for_el2 then
                E = if HaveAArch64() then MDCR_EL2.HPME else HDCR.HPME;
            else
                E = if HaveAArch64() then PMCR_EL0.E else PMCR.E;

            if HaveAArch64() then
                enabled = E == '1' && PMCNTENSET_EL0<idx> == '1';
            else
                enabled = E == '1' && PMCNTENSET<idx> == '1';

    // Event counting is allowed unless it is prohibited by any rule below
    prohibited = FALSE;

    // Event counting in Secure state is prohibited if all of:
    // * EL3 is implemented
    // * One of the following is true:
    //   - EL3 is using AArch64, MDCR_EL3.SPME == 0, and either:
    //     - FEAT_PMUv3p7 is not implemented
    //     - MDCR_EL3.MPMX == 0
    //   - EL3 is using AArch32 and SDCR.SPME == 0
    // * Executing at EL0 using AArch32 and one of the following is true:
    //   - EL3 is using AArch32 and SDER.SUNIDEN == 0
    //   - EL3 is using AArch64, EL1 is using AArch32, and SDER32_EL3.SUNIDEN == 0
    if HaveEL(EL3) && ss == SS\_Secure then
        if !ELUsingAArch32(EL3) then
            prohibited = MDCR_EL3.SPME == '0' && HavePMUv3p7() && MDCR_EL3.MPMX == '0';
        else
            prohibited = SDCR.SPME == '0';

    if prohibited && PSTATE.EL == EL0 then
        if ELUsingAArch32(EL3) then
            prohibited = SDER.SUNIDEN == '0';
        elsif ELUsingAArch32(EL1) then

```

```

        prohibited = SDER32_EL3.SUNIDEN == '0';

// Event counting at EL3 is prohibited if all of:
// * FEAT_PMUv3p7 is implemented
// * EL3 is using AArch64
// * One of the following is true:
//   - MDCR_EL3.SPME == 0
//   - PMNx is not reserved for EL2
// * MDCR_EL3.MPMX == 1
if !prohibited && HavePMUv3p7() && PSTATE.EL == EL3 && HaveAArch64() then
    prohibited = MDCR_EL3.MPMX == '1' && (MDCR_EL3.SPME == '0' || !resvd_for_el2);

// Event counting at EL2 is prohibited if all of:
// * The HPMD Extension is implemented
// * PMNx is not reserved for EL2
// * EL2 is using AArch64 and MDCR_EL2.HPMD == 1 or EL2 is using AArch32 and HDCR.HPMD == 1
if !prohibited && PSTATE.EL == EL2 && HaveHPMDExt() && !resvd_for_el2 then
    hpmd = if HaveAArch64() then MDCR_EL2.HPMD else HDCR.HPMD;
    prohibited = hpmd == '1';

// The IMPLEMENTATION DEFINED authentication interface might override software
if prohibited && !HaveNoSecurePMUDisableOverride() then
    prohibited = !ExternalSecureNoninvasiveDebugEnabled();

// Event counting might be frozen
frozen = FALSE;

// If FEAT_PMUv3p7 is implemented, event counting can be frozen
if HavePMUv3p7() then
    bit FZ;
    if resvd_for_el2 then
        FZ = if HaveAArch64() then MDCR_EL2.HPMFZO else HDCR.HPMFZO;
    else
        FZ = if HaveAArch64() then PMCR_EL0.FZO else PMCR.FZO;

    frozen = (FZ == '1') && HiLoPMUOverflow(resvd_for_el2);

    frozen = frozen || SPEFreezeOnEvent(idx);

// PMCR_EL0.DP or PMCR.DP disables the cycle counter when event counting is prohibited
if (prohibited || frozen) && idx == CYCLE_COUNTER_ID then
    dp = if HaveAArch64() then PMCR_EL0.DP else PMCR.DP;
    enabled = enabled && dp == '0';
    // Otherwise whether event counting is prohibited does not affect the cycle counter
    prohibited = FALSE;
    frozen = FALSE;

// If FEAT_PMUv3p5 is implemented, cycle counting can be prohibited.
// This is not overridden by PMCR_EL0.DP.
if HavePMUv3p5() && idx == CYCLE_COUNTER_ID then
    if HaveEL(EL3) && ss == SS_Secure then
        sccd = if HaveAArch64() then MDCR_EL3.SCCD else SDCR.SCCD;
        if sccd == '1' then
            prohibited = TRUE;

    if PSTATE.EL == EL2 then
        hccd = if HaveAArch64() then MDCR_EL2.HCCD else HDCR.HCCD;
        if hccd == '1' then
            prohibited = TRUE;

// If FEAT_PMUv3p7 is implemented, cycle counting can be prohibited at EL3.
// This is not overridden by PMCR_EL0.DP.
if HavePMUv3p7() && idx == CYCLE_COUNTER_ID then
    if PSTATE.EL == EL3 && HaveAArch64() && MDCR_EL3.MCCD == '1' then
        prohibited = TRUE;

// Event counting can be filtered by the {P, U, NSK, NSU, NSH, M, SH, RLK, RLU, RLH} bits
bits(32) filter;
case idx of
    when INSTRUCTION_COUNTER_ID

```



```

        filter = PMICFILTR_EL0<31:0>;
when CYCLE_COUNTER_ID
    filter = if HaveAArch64() then PMCCFILTR_EL0<31:0> else PMCCFILTR;
otherwise
    filter = if HaveAArch64() then PMEVTYPER_EL0[idx]<31:0> else PMEVTYPER[idx];

P    = filter<31>;
U    = filter<30>;
NSK = if HaveEL(EL3) then filter<29> else '0';
NSU = if HaveEL(EL3) then filter<28> else '0';
NSH = if HaveEL(EL2) then filter<27> else '0';
M    = if HaveEL(EL3) && HaveAArch64() then filter<26> else '0';
SH    = if HaveEL(EL3) && HaveSecureEL2Ext() then filter<24> else '0';
RLK = if HaveRME() then filter<22> else '0';
RLU = if HaveRME() then filter<21> else '0';
RLH = if HaveRME() then filter<20> else '0';

ss = CurrentSecurityState();
case PSTATE.EL of
when EL0
    case ss of
        when SS_NonSecure filtered = U != NSU;
        when SS_Secure   filtered = U == '1';
        when SS_Realm     filtered = U != RLU;
when EL1
    case ss of
        when SS_NonSecure filtered = P != NSK;
        when SS_Secure   filtered = P == '1';
        when SS_Realm     filtered = P != RLK;
when EL2
    case ss of
        when SS_NonSecure filtered = NSH == '0';
        when SS_Secure   filtered = NSH == SH;
        when SS_Realm     filtered = NSH == RLH;
when EL3
    if HaveAArch64() then
        filtered = M != P;
    else
        filtered = P == '1';

return !debug && enabled && !prohibited && !filtered && !frozen;

```

### Library pseudocode for shared/debug/pmu/GetNumEventCounters

```

// GetNumEventCounters()
// =====
// Returns the number of event counters implemented. This is indicated to software at the
// highest Exception level by PMCR.N in AArch32 state, and PMCR_EL0.N in AArch64 state.

integer GetNumEventCounters()
    return integer IMPLEMENTATION_DEFINED "Number of event counters";

```

### Library pseudocode for shared/debug/pmu/HasElapsed64Cycles

```

// HasElapsed64Cycles()
// =====
// Returns TRUE if 64 cycles have elapsed between the last count, and FALSE otherwise.

boolean HasElapsed64Cycles();

```

## Library pseudocode for shared/debug/pmu/HiLoPMUOverflow

```
// HiLoPMUOverflow()
// =====

boolean HiLoPMUOverflow(boolean resvd_for_el2)
    boolean check_cnten = FALSE;
    boolean check_e      = FALSE;
    boolean check_inten  = FALSE;
    boolean include_lo   = !resvd_for_el2;
    boolean include_hi   = resvd_for_el2;
    boolean exclude_cyc  = FALSE;
    boolean exclude_sync = FALSE;

    boolean overflow = PMUOverflowCondition(check_e, check_cnten, check_inten,
                                              include_hi, include_lo,
                                              exclude_cyc, exclude_sync);

    return overflow;
```

## Library pseudocode for shared/debug/pmu/INSTRUCTION\_COUNTER\_ID

```
constant integer INSTRUCTION_COUNTER_ID = 32;
```

## Library pseudocode for shared/debug/pmu/IncrementInstructionCounter

```
// IncrementInstructionCounter()
// =====
// Increment the instruction counter and possibly set overflow bits.

IncrementInstructionCounter(integer increment)
    if CountPMUEvents(INSTRUCTION_COUNTER_ID) then
        integer old_value = UInt(PMICNTR_EL0);
        integer new_value = old_value + increment;
        PMICNTR_EL0      = new_value<63:0>;

        // The effective value of PMCR_EL0.LP is '1' for the instruction counter
        if old_value<64> != new_value<64> then
            PMOVSSET_EL0.F0 = '1';
            PMOVSCLR_EL0.F0 = '1';

    return;
```

## Library pseudocode for shared/debug/pmu/PMUCaptureEvent

```
// PMUCaptureEvent()
// =====
// If permitted and enabled, generate a PMU snapshot Capture event.

PMUCaptureEvent()
    assert HaveEL\(EL3\) && IsFeatureImplemented(FEAT_PMUv3_SS) && HaveAArch64\(\);
    boolean debug_state = Halted\(\);

    if !PMUCaptureEventAllowed\(\) then
        // Indicate a Capture event in progress
        PMSSCR_EL1.<NC,SS> = '10';
        return;

    for idx = 0 to GetNumEventCounters\(\) - 1
        PMEVCNTSVR_EL1[idx] = PMEVCNTR_EL0[idx];

    PMCCNTSVR_EL1 = PMCCNTR_EL0;

    if HavePMUv3ICNTR\(\) then
        PMICNTSVR_EL1 = PMICNTR_EL0;

    if IsFeatureImplemented(FEAT_PCSRv8p9) && PMPCSTL_SS == '1' then
        if pc_sample.valid && !debug_state then
            SetPCSample\(\);
        else
            SetPCSRUnknown\(\);

    if (HaveBRBExt\(\) && BranchRecordAllowed(PSTATE.EL) && BRBCR_EL1.FZPSS == '1' &&
        (!HaveEL\(EL2\) || BRBCR_EL2.FZPSS == '1')) then
        BRBEFreeze\(\);

    // Indicate a successful Capture event
    PMSSCR_EL1.<NC,SS> = '00';
    if !debug_state || ConstrainUnpredictableBool(Unpredictable\_PMUSNAPSHOTEVENT) then
        PMUEvent(PMU_EVENT_PMU_SNAPSHOT);

    return;
```

## Library pseudocode for shared/debug/pmu/PMUCaptureEventAllowed

```
// PMUCaptureEventAllowed()
// =====
// Returns TRUE if PMU Capture events are allowed, and FALSE otherwise.

boolean PMUCaptureEventAllowed()
    if !IsFeatureImplemented(FEAT_PMUv3_SS) || OSLockStatus\(\) || !HaveAArch64\(\) then
        return FALSE;
    if HaveEL\(EL3\) && MDCR_EL3.PMSSE != '01' then
        return MDCR_EL3.PMSSE == '11';
    elsif EL2Enabled\(\) && ELUsingAArch32\(EL2\) then
        return FALSE;
    elsif EL2Enabled\(\) && MDCR_EL2.PMSSE != '01' then
        return MDCR_EL2.PMSSE == '11';
    elsif ELUsingAArch32\(EL1\) then
        return FALSE;
    else
        bits(2) pmsse_el1 = PMECCR_EL1.SSE;
        if pmsse_el1 == '01' then // Reserved value
            Constraint c;
            (c, pmsse_el1) = ConstrainUnpredictableBits(Unpredictable\_RESPMSSE, 2);
            assert c IN {Constraint\_DISABLED, Constraint\_UNKNOWN};
            if c == Constraint\_DISABLED then pmsse_el1 = '00';
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value
        return pmsse_el1 == '11';
```

## Library pseudocode for shared/debug/pmu/PMUCaptureEventEnabled

```
// PMUCaptureEventEnabled()
// =====
// Returns TRUE if PMU Capture events are enabled, and FALSE otherwise.

boolean PMUCaptureEventEnabled()
    if !IsFeatureImplemented(FEAT_PMUv3_SS) || !HaveAArch64() then
        return FALSE;
    if HaveEL(EL3) && MDCR_EL3.PMSSE != '01' then
        return MDCR_EL3.PMSSE IN {'1x'};
    elsif EL2Enabled() && ELUsingAArch32(EL2) then
        return FALSE;
    elsif EL2Enabled() && MDCR_EL2.PMSSE != '01' then
        return MDCR_EL2.PMSSE IN {'1x'};
    elsif ELUsingAArch32(EL1) then
        return FALSE;
    else
        bits(2) pmsse_el1 = PMECR_EL1.SSE;
        if pmsse_el1 == '01' then // Reserved value
            Constraint c;
            (c, pmsse_el1) = ConstrainUnpredictableBits(Unpredictable_RESPMSSE, 2);
            assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
            if c == Constraint_DISABLED then pmsse_el1 = '00';
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value
        return pmsse_el1 IN {'1x'};
```

## Library pseudocode for shared/debug/pmu/PMUCountValue

```
// PMUCountValue()
// =====
// Implements the PMU threshold function, if implemented.
// Returns the value to increment event counter 'n' by.
// 'Vb' is the base value of the event that event counter 'n' is configured to count.

integer PMUCountValue(integer n, integer Vb)
  if !HavePMUv3TH() || !HaveAArch64() then
    return Vb;

  integer T = UInt(PMEVTYPER_EL0[n].TH);
  boolean Vc;

  case PMEVTYPER_EL0[n].TC<2:1> of
    when '00' Vc = (Vb != T);      // Disabled or not-equal
    when '01' Vc = (Vb == T);      // Equals
    when '10' Vc = (Vb >= T);      // Greater-than-or-equal
    when '11' Vc = (Vb < T);       // Less-than

  integer Vt;
  if PMEVTYPER_EL0[n].TC<0> == '0' then
    Vt = (if Vc then Vb else 0);    // Count values
  else
    Vt = (if Vc then 1 else 0);     // Count matches

  integer v;
  if HavePMUv3EDGE() && PMEVTYPER_EL0[n].TE == '1' then
    Vp = PMULastThresholdValue[n];

    tc = PMEVTYPER_EL0[n].TC<1:0>;
    // Check for reserved case
    if tc == '00' then
      Constraint c;
      (c, tc) = ConstrainUnpredictableBits(Unpredictable\_RESTC, 2);
      if c == Constraint\_DISABLED then tc = '00';
      // Otherwise the value returned by ConstrainUnpredictableBits
      // must be a not-reserved value.

    case tc of
      when '00' v = Vt;              // Reserved - treat as disabled
      when '10' v = (if Vp != Vc then 1 else 0); // Both edges
      when 'x1' v = (if !Vp && Vc then 1 else 0); // Single edge
    else
      v = Vt;

  PMULastThresholdValue[n] = Vc;

  return v;
```

## Library pseudocode for shared/debug/pmu/PMUCounterIsHyp

```
// PMUCounterIsHyp()
// =====
// Returns TRUE if a counter is reserved for use by EL2, FALSE otherwise.

boolean PMUCounterIsHyp(integer n)
    if n == INSTRUCTION\_COUNTER\_ID then return FALSE;
    if n == CYCLE\_COUNTER\_ID then return FALSE;

    boolean resvd_for_el2;
    if HaveEL\(EL2\) then // Software can reserve some event counters for EL2
        bits(5) hpmn_bits = if HaveAArch64\(\) then MDCR_EL2.HPMN else HDCR.HPMN;
        resvd_for_el2 = n >= UInt\(hpmn\_bits\);
        if UInt\(hpmn\_bits\) > GetNumEventCounters\(\) || (!HaveFeatHPMN0\(\) && IsZero\(hpmn\_bits\)) then
            resvd_for_el2 = ConstrainUnpredictableBool\(Unpredictable\_CounterReservedForEL2\);
    else
        resvd_for_el2 = FALSE;

    return resvd_for_el2;
```

## Library pseudocode for shared/debug/pmu/PMUCounterMask

```
// PMUCounterMask()
// =====
// Return bitmask of accessible PMU counters.

bits(64) PMUCounterMask()
    integer n;
    if UsingAArch32\(\) then
        n = AArch32.GetNumEventCountersAccessible\(\);
    else
        n = AArch64.GetNumEventCountersAccessible\(\);

    mask = ZeroExtend\(Ones\(n\), 64\);
    mask<CYCLE\_COUNTER\_ID> = '1';
    if HaveAArch64\(\) && HavePMUv3ICNTR\(\) then mask<INSTRUCTION\_COUNTER\_ID> = '1';
    return mask;
```

## Library pseudocode for shared/debug/pmu/PMUEvent

```
constant bits(16) PMU_EVENT_PMU_SNAPSHOT = 0x8127<15:0>;
constant bits(16) PMU_EVENT_SAMPLE_FEED_BR = 0x812A<15:0>;
constant bits(16) PMU_EVENT_SAMPLE_FEED_LD = 0x812B<15:0>;
constant bits(16) PMU_EVENT_SAMPLE_FEED_ST = 0x812C<15:0>;
constant bits(16) PMU_EVENT_SAMPLE_FEED_OP = 0x812D<15:0>;
constant bits(16) PMU_EVENT_SAMPLE_FEED_EVENT = 0x812E<15:0>;
constant bits(16) PMU_EVENT_SAMPLE_FEED_LAT = 0x812F<15:0>;
constant bits(16) PMU_EVENT_DSNP_HIT_RD = 0x8194<15:0>;
constant bits(16) PMU_EVENT_L1D_CACHE_HITM_RD = 0x8214<15:0>;
constant bits(16) PMU_EVENT_L2D_CACHE_HITM_RD = 0x8215<15:0>;
constant bits(16) PMU_EVENT_L3D_CACHE_HITM_RD = 0x8216<15:0>;
constant bits(16) PMU_EVENT_LL_CACHE_HITM_RD = 0x8217<15:0>;
constant bits(16) PMU_EVENT_L1D_LFB_HIT_RD = 0x8244<15:0>;
constant bits(16) PMU_EVENT_L2D_LFB_HIT_RD = 0x8245<15:0>;
constant bits(16) PMU_EVENT_L3D_LFB_HIT_RD = 0x8246<15:0>;
constant bits(16) PMU_EVENT_LL_LFB_HIT_RD = 0x8247<15:0>;

// PMUEvent()
// =====
// Generate a PMU event. By default, increment by 1.

PMUEvent(bits(16) pmuevent)
    PMUEvent(pmuevent, 1);

// PMUEvent()
// =====
// Accumulate a PMU Event.

PMUEvent(bits(16) pmuevent, integer increment)
    if SPESampleInFlight then
        SPEEvent(pmuevent);
    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            PMUEvent(pmuevent, increment, idx);

    if HaveAArch64() && HavePMUv3ICNTR() && pmuevent == PMU_EVENT_INST_RETIRED then
        IncrementInstructionCounter(increment);

// PMUEvent()
// =====
// Accumulate a PMU Event for a specific event counter.

PMUEvent(bits(16) pmuevent, integer increment, integer idx)
    if !HavePMUv3() then
        return;

    if UsingAArch32() then
        if PMEVTYPEPER[idx].evtCount == pmuevent then
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;
    else
        if PMEVTYPEPER_EL0[idx].evtCount == pmuevent then
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;
```





```

// PMUOverflowCondition()
// =====
// Checks for PMU overflow under certain parameter conditions
// If 'check_e' is TRUE, then check the applicable one of PMCR_EL0.E and MDCR_EL2.HPME.
// If 'check_cnten' is TRUE, then check the applicable PMCNTENCLR_EL0 bit.
// If 'check_cnten' is TRUE, then check the applicable PMINTENCLR_EL1 bit.
// If 'include_lo' is TRUE, then check counters in the set [0..(HPMN-1)], CCNTR
// and ICNTR, unless excluded by other flags.
// If 'include_hi' is TRUE, then check counters in the set [HPMN..(N-1)].
// If 'exclude_cyc' is TRUE, then CCNTR is NOT checked.
// If 'exclude_sync' is TRUE, then counters in synchronous mode are NOT checked.

boolean PMUOverflowCondition(boolean check_e, boolean check_cnten,
                             boolean check_inten,
                             boolean include_hi, boolean include_lo,
                             boolean exclude_cyc, boolean exclude_sync)
integer counters = GetNumEventCounters();

bits(64) ovsf;

if HaveAArch64() then
    ovsf = PMOVSLR_EL0;

    // Remove unimplemented counters - these fields are RES0
    ovsf<63:33> = Zeros(31);

    if !HavePMUv3ICNTR() then
        ovsf<INSTRUCTION\_COUNTER\_ID> = '0';
else
    ovsf = ZeroExtend(PMOVSR, 64);

if counters < 31 then
    ovsf<30:counters> = Zeros(31-counters);

for idx = 0 to counters - 1
    bit E;

    boolean is_hyp = PMUCounterIsHyp(idx);
    if HaveAArch64() then
        E = (if is_hyp then MDCR_EL2.HPME else PMCR_EL0.E);

        if exclude_sync then
            bit sync = (PMCNTENCLR_EL0[idx] AND PMEVTYPER_EL0[idx].SYNC);
            ovsf<idx> = ovsf<idx> AND NOT sync;
        else
            E = (if is_hyp then HDCR.HPME else PMCR.E);

    if check_e then
        ovsf<idx> = ovsf<idx> AND E;

    if (!is_hyp && !include_lo) || (is_hyp && !include_hi) then
        ovsf<idx> = '0';

// Cycle counter
if exclude_cyc || !include_lo then
    ovsf<CYCLE\_COUNTER\_ID> = '0';

if check_e then
    ovsf<CYCLE\_COUNTER\_ID> = ovsf<CYCLE\_COUNTER\_ID> AND PMCR_EL0.E;

// Instruction counter
if HaveAArch64() && HavePMUv3ICNTR() then
    if !include_lo then
        ovsf<INSTRUCTION\_COUNTER\_ID> = '0';
    if exclude_sync then
        bit sync = (PMCNTENCLR_EL0.F0 AND PMICFILTR_EL0.SYNC);
        ovsf<INSTRUCTION\_COUNTER\_ID> = ovsf<INSTRUCTION\_COUNTER\_ID> AND NOT sync;

    if check_e then
        ovsf<INSTRUCTION\_COUNTER\_ID> = ovsf<INSTRUCTION\_COUNTER\_ID> AND PMCR_EL0.E;

```

```

if check_cnten then
    bits(64) cnten = if HaveAArch64() then PMCNTENCLR_EL0 else ZeroExtend(PMCNTENCLR, 64);
    ovsf = ovsf AND cnten;

if check_inten then
    bits(64) inten = if HaveAArch64() then PMINTENCLR_EL1 else ZeroExtend(PMINTENCLR, 64);
    ovsf = ovsf AND inten;

return !IsZero(ovsf);

```

### Library pseudocode for shared/debug/samplebasedprofiling/CreatePCSample

```

// CreatePCSample()
// =====

CreatePCSample()
    // In a simple sequential execution of the program, CreatePCSample is executed each time the PE
    // executes an instruction that can be sampled. An implementation is not constrained such that
    // reads of EDPCSRlo return the current values of PC, etc.

    if IsFeatureImplemented(FEAT_PCSRv8p9) && PCSRsSuspended() then return;
    pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
    pc_sample.pc = ThisInstrAddr(64);
    pc_sample.el = PSTATE.EL;
    pc_sample.rw = if UsingAArch32() then '0' else '1';
    pc_sample.ss = CurrentSecurityState();
    pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR else CONTEXTIDR_EL1<31:0>;
    pc_sample.has_el2 = PSTATE.EL != EL3 && EL2Enabled();

    if pc_sample.has_el2 then
        if ELUsingAArch32(EL2) then
            pc_sample.vmid = ZeroExtend(VTTBR.VMID, 16);
        elsif !Have16bitVMID() || VTCR_EL2.VS == '0' then
            pc_sample.vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        else
            pc_sample.vmid = VTTBR_EL2.VMID;
        if (HaveVirtHostExt() || HaveV82Debug()) && !ELUsingAArch32(EL2) then
            pc_sample.contextidr_el2 = CONTEXTIDR_EL2<31:0>;
        else
            pc_sample.contextidr_el2 = bits(32) UNKNOWN;
        pc_sample.el0h = PSTATE.EL == EL0 && IsInHost();
    return;

```

### Library pseudocode for shared/debug/samplebasedprofiling/PCSRsSuspended

```

// PCSRsSuspended()
// =====
// Returns TRUE if PC Sample-based Profiling is suspended, and FALSE otherwise.

boolean PCSRsSuspended()
    if PMPCSTL.IMP == '1' then
        return PMPCSTL.EN == '0';
    else
        return boolean IMPLEMENTATION_DEFINED "PCSR is suspended";

```

## Library pseudocode for shared/debug/samplebasedprofiling/PCSample

```
PCSample pc_sample;

// PCSample
// =====

type PCSample is (
    boolean valid,
    bits(64) pc,
    bits(2) el,
    bit rw,
    SecurityState ss,
    boolean has_el2,
    bits(32) contextidr,
    bits(32) contextidr_el2,
    boolean el0h,
    bits(16) vmid
)
```

## Library pseudocode for shared/debug/samplebasedprofiling/Read\_EDPCSRlo

```
// Read_EDPCSRlo()
// =====

bits(32) Read_EDPCSRlo(boolean memory_mapped)

    if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return bits(32) UNKNOWN;

    // The Software lock is OPTIONAL.
    update = !memory_mapped || EDLSR.SLK == '0'; // Software locked: no side-effects
    bits(32) sample;
    if pc_sample.valid then
        sample = pc_sample.pc<31:0>;
        if update then
            if HaveVirtHostExt() && EDSCR.SC2 == '1' then
                EDPCSRhi.PC = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
                EDPCSRhi.EL = pc_sample.el;
                EDPCSRhi.NS = (if pc_sample.ss == SS_Secure then '0' else '1');
            else
                EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else pc_sample.pc<63:32>);
            EDCIDSR = pc_sample.contextidr;
            if (HaveVirtHostExt() || HaveV82Debug()) && EDSCR.SC2 == '1' then
                EDVIDSR = (if pc_sample.has_el2 then pc_sample.contextidr_el2
                    else bits(32) UNKNOWN);
            else
                EDVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0}
                    then pc_sample.vmid else Zeros(16));
                EDVIDSR.NS = (if pc_sample.ss == SS_Secure then '0' else '1');
                EDVIDSR.E2 = (if pc_sample.el == EL2 then '1' else '0');
                EDVIDSR.E3 = (if pc_sample.el == EL3 then '1' else '0') AND pc_sample.rw;
                // The conditions for setting HV are not specified if PCSRhi is zero.
                // An example implementation may be "pc_sample.rw".
                EDVIDSR.HV = (if !IsZero(EDPCSRhi) then '1'
                    else bit IMPLEMENTATION_DEFINED "0 or 1");
        else
            sample = Ones(32);
            if update then
                EDPCSRhi = bits(32) UNKNOWN;
                EDCIDSR = bits(32) UNKNOWN;
                EDVIDSR = bits(32) UNKNOWN;

    return sample;
```

## Library pseudocode for shared/debug/samplebasedprofiling/Read\_PMPCSR

```
// Read_PMPCSR()
// =====

bits(64) Read_PMPCSR(boolean memory_mapped)
    if EDPRSR<6:5,0> != '001' then
        IMPLEMENTATION_DEFINED "generate error response";
        return bits(64) UNKNOWN;

    // The Software lock is OPTIONAL.
    update = !memory_mapped || PMLSR.SLK == '0';           // Software locked: no side-effects

    if IsFeatureImplemented(FEAT_PCSRv8p9) && update then
        if IsFeatureImplemented(FEAT_PMUv3_SS) && PMPCSCTL.SS == '1' then
            update = FALSE;
        elsif PMPCSCTL.<IMP,EN> == '10' || (PMPCSCTL.IMP == '0' && PCSRsSuspended\(\)) then
            pc_sample.valid = FALSE;
            SetPCSRActive\(\);

    if pc_sample.valid then
        if update then SetPCSample\(\);
        return PMPCSR;
    else
        if update then SetPCSRUnknown\(\);
        return (bits(32) UNKNOWN : Ones(32));
```

## Library pseudocode for shared/debug/samplebasedprofiling/SetPCSRActive

```
// SetPCSRActive()
// =====
// Sets PC Sample-based Profiling to active state.

SetPCSRActive()
    if PMPCSCTL.IMP == '1' then
        PMPCSCTL.EN = '1';
    // If PMPCSCTL.IMP reads as `0b0`, then PMPCSCTL.EN is RES0, and it is
    // IMPLEMENTATION_DEFINED whether PCSR is suspended or active at reset.
```

## Library pseudocode for shared/debug/samplebasedprofiling/SetPCSRUnknown

```
// SetPCSRUnknown()
// =====
// Sets the PC sample registers to UNKNOWN values because PC sampling
// is prohibited.

SetPCSRUnknown()
    PMPCSR<31:0> = Ones(32);
    PMPCSR<55:32> = bits(24) UNKNOWN;
    PMPCSR.EL = bits(2) UNKNOWN;
    PMPCSR.NS = bit UNKNOWN;

    PMCID1SR = bits(32) UNKNOWN;
    PMCID2SR = bits(32) UNKNOWN;

    PMVIDSR.VMID = bits(16) UNKNOWN;

    return;
```

## Library pseudocode for shared/debug/samplebasedprofiling/SetPCSample

```
// SetPCSample()
// =====
// Sets the PC sample registers to the appropriate sample values.

SetPCSample()
    PMPCSR<31:0> = pc_sample.pc<31:0>;
    PMPCSR<55:32> = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
    PMPCSR.EL = pc_sample.el;
    if HaveRME() then
        case pc_sample.ss of
            when SS\_Secure
                PMPCSR.NSE = '0'; PMPCSR.NS = '0';
            when SS\_NonSecure
                PMPCSR.NSE = '0'; PMPCSR.NS = '1';
            when SS\_Root
                PMPCSR.NSE = '1'; PMPCSR.NS = '0';
            when SS\_Realm
                PMPCSR.NSE = '1'; PMPCSR.NS = '1';
        else
            PMPCSR.NS = (if pc_sample.ss == SS\_Secure then '0' else '1');

    PMCID1SR = pc_sample.contextidr;
    PMCID2SR = if pc_sample.has_el2 then pc_sample.contextidr_el2 else bits(32) UNKNOWN;

    PMVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0} && !pc_sample.el0h
        then pc_sample.vmid else bits(16) UNKNOWN);

    return;
```

## Library pseudocode for shared/debug/softwarestep/CheckSoftwareStep

```
// CheckSoftwareStep()
// =====
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

    // Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
    // AArch32 state. However, because Software Step is only active when the debug target Exception
    // level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
    step_enabled = (!ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() &&
        MDSCR_EL1.SS == '1');
    active_pending = step_enabled && PSTATE.SS == '0';    // active-pending
    if active_pending then
        AArch64.SoftwareStepException();
    ShouldAdvanceSS = TRUE;
    return;
```

## Library pseudocode for shared/debug/softwarestep/DebugExceptionReturnSS

```
// DebugExceptionReturnSS()
// =====
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(N) spsr)
    assert Halted\(\) || Restarting\(\) || PSTATE.EL != EL0;

    boolean enabled_at_source;
    if Restarting\(\) then
        enabled_at_source = FALSE;
    elseif UsingAArch32\(\) then
        enabled_at_source = AArch32.GenerateDebugExceptions\(\);
    else
        enabled_at_source = AArch64.GenerateDebugExceptions\(\);

    boolean valid;
    bits(2) dest_el;
    if IllegalExceptionReturn(spsr) then
        dest_el = PSTATE.EL;
    else
        (valid, dest_el) = ELFromSPSR(spsr); assert valid;

    dest_ss = SecurityStateAtEL(dest_el);
    bit mask;
    boolean enabled_at_dest;
    dest_using_32 = (if dest_el == EL0 then spsr<4> == '1' else ELUsingAArch32(dest_el));
    if dest_using_32 then
        enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest_el, dest_ss);
    else
        mask = spsr<9>;
        enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest_el, dest_ss, mask);

    ELd = DebugTargetFrom(dest_ss);
    bit SS_bit;
    if !ELUsingAArch32(ELd) && MDSCR_EL1.SS == '1' && !enabled_at_source && enabled_at_dest then
        SS_bit = spsr<21>;
    else
        SS_bit = '0';

    return SS_bit;
```

## Library pseudocode for shared/debug/softwarestep/SSAdvance

```
// SSAdvance()
// =====
// Advance the Software Step state machine.

SSAdvance()

    // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
    // current Software Step state machine. However, this check is made to illustrate that the
    // processor only needs to consider advancing the state machine from the active-not-pending
    // state.
    if !ShouldAdvanceSS then return;
    target = DebugTarget();
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';
    if active_not_pending then PSTATE.SS = '0';
    ShouldAdvanceSS = FALSE;
    return;
```

## Library pseudocode for shared/debug/softwarestep/SoftwareStep\_DidNotStep

```
// SoftwareStep_DidNotStep()
// =====
// Returns TRUE if the previously executed instruction was executed in the
// inactive state, that is, if it was not itself stepped.
// Might return TRUE or FALSE if the previously executed instruction was an ISB
// or ERET executed in the active-not-pending state, or if another exception
// was taken before the Software Step exception. Returns FALSE otherwise,
// indicating that the previously executed instruction was executed in the
// active-not-pending state, that is, the instruction was stepped.

boolean SoftwareStep_DidNotStep();
```

## Library pseudocode for shared/debug/softwarestep/SoftwareStep\_SteppedEX

```
// SoftwareStep_SteppedEX()
// =====
// Returns a value that describes the previously executed instruction. The
// result is valid only if SoftwareStep_DidNotStep() returns FALSE.
// Might return TRUE or FALSE if the instruction was an AArch32 LDREX or LDAEX
// that failed its condition code test. Otherwise returns TRUE if the
// instruction was a Load-Exclusive class instruction, and FALSE if the
// instruction was not a Load-Exclusive class instruction.
boolean SoftwareStep_SteppedEX();
```

## Library pseudocode for shared/exceptions/exceptions/ConditionSyndrome

```
// ConditionSyndrome()
// =====
// Return CV and COND fields of instruction syndrome

bits(5) ConditionSyndrome()

    bits(5) syndrome;

    if UsingAArch32\(\) then
        cond = AArch32.CurrentCond\(\);
        if PSTATE.T == '0' then // A32
            syndrome<4> = '1';
            // A conditional A32 instruction that is known to pass its condition code check
            // can be presented either with COND set to 0xE, the value for unconditional, or
            // the COND value held in the instruction.
            if ConditionHolds(cond) && ConstrainUnpredictableBool(Unpredictable\_ESRCONDPASS) then
                syndrome<3:0> = '1110';
            else
                syndrome<3:0> = cond;
        else // T32
            // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
            // * CV set to 0 and COND is set to an UNKNOWN value
            // * CV set to 1 and COND is set to the condition code for the condition that
            // applied to the instruction.
            if boolean IMPLEMENTATION_DEFINED "Condition valid for trapped T32" then
                syndrome<4> = '1';
                syndrome<3:0> = cond;
            else
                syndrome<4> = '0';
                syndrome<3:0> = bits(4) UNKNOWN;
    else
        syndrome<4> = '1';
        syndrome<3:0> = '1110';

    return syndrome;
```

## Library pseudocode for shared/exceptions/exceptions/Exception

```
// Exception
// =====
// Classes of exception.

enumeration Exception {
    Exception_Uncategorized,    // Uncategorized or unknown reason
    Exception_WFxTrap,          // Trapped WFI or WFE instruction
    Exception_CP15RTTTrap,      // Trapped AArch32 MCR or MRC access, coproc=0b111
    Exception_CP15RRTTrap,      // Trapped AArch32 MCRR or MRRC access, coproc=0b1111
    Exception_CP14RTTTrap,      // Trapped AArch32 MCR or MRC access, coproc=0b1110
    Exception_CP14DTTTrap,      // Trapped AArch32 LDC or STC access, coproc=0b1110
    Exception_CP14RRTTrap,      // Trapped AArch32 MRRC access, coproc=0b1110
    Exception_AdvSIMDFPAccessTrap, // HCPTR-trapped access to SIMD or FP
    Exception_FPIDTrap,         // Trapped access to SIMD or FP ID register
    Exception_LDST64BTrap,      // Trapped access to ST64BV, ST64BV0, ST64B and LD64B
    // Trapped BXJ instruction not supported in Armv8
    Exception_PACTrap,          // Trapped invalid PAC use
    Exception_IllegalState,     // Illegal Execution state
    Exception_SupervisorCall,   // Supervisor Call
    Exception_HypervisorCall,   // Hypervisor Call
    Exception_MonitorCall,      // Monitor Call or Trapped SMC instruction
    Exception_SystemRegisterTrap, // Trapped MRS or MSR System register access
    Exception_ERetTrap,         // Trapped invalid ERET use
    Exception_InstructionAbort,  // Instruction Abort or Prefetch Abort
    Exception_PCAlignment,      // PC alignment fault
    Exception_DataAbort,        // Data Abort
    Exception_NV2DataAbort,     // Data abort at EL1 reported as being from EL2
    Exception_PACFail,          // PAC Authentication failure
    Exception_SPAlignment,      // SP alignment fault
    Exception_FPTrappedException, // IEEE trapped FP exception
    Exception_SError,           // SError interrupt
    Exception_Breakpoint,       // (Hardware) Breakpoint
    Exception_SoftwareStep,     // Software Step
    Exception_Watchpoint,       // Watchpoint
    Exception_NV2Watchpoint,    // Watchpoint at EL1 reported as being from EL2
    Exception_SoftwareBreakpoint, // Software Breakpoint Instruction
    Exception_VectorCatch,      // AArch32 Vector Catch
    Exception_IRQ,              // IRQ interrupt
    Exception_SVEAccessTrap,    // HCPTR trapped access to SVE
    Exception_SMEAccessTrap,    // HCPTR trapped access to SME
    Exception_TSTARTAccessTrap, // Trapped TSTART access
    Exception_GPC,              // Granule protection check
    Exception_BranchTarget,     // Branch Target Identification
    Exception_MemCpyMemSet,     // Exception from a CPY* or SET* instruction
    Exception_GCSFail,          // GCS Exceptions
    Exception_PMU,              // PMU exception
    Exception_SystemRegister128Trap, // Trapped MRRS or MSRR System register or SYSP access
    Exception_FIQ;              // FIQ interrupt
}
```

## Library pseudocode for shared/exceptions/exceptions/ExceptionRecord

```
// ExceptionRecord
// =====

type ExceptionRecord is (
    Exception    exceptype,    // Exception class
    bits(25)    syndrome,      // Syndrome record
    bits(24)    syndrome2,     // Syndrome record
    FullAddress paddress,      // Physical fault address
    bits(64)    vaddress,      // Virtual fault address
    boolean     ipavalid,       // Validity of Intermediate Physical fault address
    boolean     pavalid,       // Validity of Physical fault address
    bit         NS,            // Intermediate Physical fault address space
    bits(56)    ipaddress,     // Intermediate Physical fault address
    boolean     trappedsyscallinst) // Trapped SVC or SMC instruction
}
```



## Library pseudocode for shared/exceptions/exceptions/ExceptionSyndrome

```
// ExceptionSyndrome()
// =====
// Return a blank exception syndrome record for an exception of the given type.

ExceptionRecord ExceptionSyndrome(Exception exceptype)

    ExceptionRecord r;

    r.exceptype = exceptype;

    // Initialize all other fields
    r.syndrome = Zeros(25);
    r.syndrome2 = Zeros(24);
    r.vaddress = Zeros(64);
    r.ipavalid = FALSE;
    r.NS = '0';
    r.ipaddress = Zeros(56);
    r.paddress.paspace = PASpace UNKNOWN;
    r.paddress.address = bits(56) UNKNOWN;
    r.trappedsyscallinst = FALSE;
    return r;
```

## Library pseudocode for shared/exceptions/traps/Undefined

```
// Undefined()
// =====

Undefined()
    if UsingAArch32() then
        AArch32.Undefined();
    else
        AArch64.Undefined();
```

## Library pseudocode for shared/functions/aborts/EncodeLDFSC

```
// EncodeLDFSC()
// =====
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault statuscode, integer level)
    bits(6) result;

    // 128-bit descriptors will start from level -2 for 4KB to resolve bits IA[55:51]
    if level == -2 then
        assert Have56BitPAExt();
        case statuscode of
            when Fault AddressSize                result = '101100';
            when Fault Translation                result = '101010';
            when Fault SyncExternalOnWalk          result = '010010';
            when Fault SyncParityOnWalk            result = '011010'; assert !HaveRASExt();
            when Fault GPCFOnWalk                  result = '100010';
            otherwise                               Unreachable();
        return result;

    if level == -1 then
        assert Have52BitIPAAndPASpaceExt();
        case statuscode of
            when Fault AddressSize                result = '101001';
            when Fault Translation                result = '101011';
            when Fault SyncExternalOnWalk          result = '010011';
            when Fault SyncParityOnWalk            result = '011011'; assert !HaveRASExt();
            when Fault GPCFOnWalk                  result = '100011';
            otherwise                               Unreachable();

        return result;
    case statuscode of
        when Fault AddressSize                result = '0000':level<1:0>; assert level IN {0,1,2,3};
        when Fault AccessFlag                 result = '0010':level<1:0>; assert level IN {0,1,2,3};
        when Fault Permission                 result = '0011':level<1:0>; assert level IN {0,1,2,3};
        when Fault Translation                 result = '0001':level<1:0>; assert level IN {0,1,2,3};
        when Fault SyncExternal                result = '010000';
        when Fault SyncExternalOnWalk           result = '0101':level<1:0>; assert level IN {0,1,2,3};
        when Fault SyncParity                  result = '011000';
        when Fault SyncParityOnWalk             result = '0111':level<1:0>; assert level IN {0,1,2,3};
        when Fault AsyncParity                 result = '011001';
        when Fault AsyncExternal               result = '010001'; assert UsingAArch32();
        when Fault TagCheck                   result = '010001'; assert HaveMTE2Ext();
        when Fault Alignment                  result = '100001';
        when Fault Debug                      result = '100010';
        when Fault GPCFOnWalk                  result = '1001':level<1:0>; assert level IN {0,1,2,3};
        when Fault GPCFOnOutput                result = '101000';
        when Fault TLBConflict                 result = '110000';
        when Fault HWUpdateAccessFlag          result = '110001';
        when Fault Lockdown                   result = '110100'; // IMPLEMENTATION DEFINED
        when Fault Exclusive                  result = '110101'; // IMPLEMENTATION DEFINED
        otherwise                               Unreachable();

    return result;
```

## Library pseudocode for shared/functions/aborts/IPAValid

```
// IPAValid()
// =====
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.statuscode != Fault\_None;

    if fault.gpcf.gpcf != GPCF\_None then
        return fault.secondstage;
    elsif fault.s2fslwalk then
        return fault.statuscode IN {
            Fault\_AccessFlag,
            Fault\_Permission,
            Fault\_Translation,
            Fault\_AddressSize
        };
    elsif fault.secondstage then
        return fault.statuscode IN {
            Fault\_AccessFlag,
            Fault\_Translation,
            Fault\_AddressSize
        };
    else
        return FALSE;
```

## Library pseudocode for shared/functions/aborts/IsAsyncAbort

```
// IsAsyncAbort()
// =====
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

boolean IsAsyncAbort(Fault statuscode)
    assert statuscode != Fault\_None;

    return (statuscode IN {Fault\_AsyncExternal, Fault\_AsyncParity});

// IsAsyncAbort()
// =====

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.statuscode);
```

## Library pseudocode for shared/functions/aborts/IsDebugException

```
// IsDebugException()
// =====

boolean IsDebugException(FaultRecord fault)
    assert fault.statuscode != Fault\_None;
    return fault.statuscode == Fault\_Debug;
```

## Library pseudocode for shared/functions/aborts/IsExternalAbort

```
// IsExternalAbort()
// =====
// Returns TRUE if the abort currently being processed is an External abort and FALSE otherwise.

boolean IsExternalAbort(Fault statuscode)
    assert statuscode != Fault\_None;

    return (statuscode IN {
        Fault\_SyncExternal,
        Fault\_SyncParity,
        Fault\_SyncExternalOnWalk,
        Fault\_SyncParityOnWalk,
        Fault\_AsyncExternal,
        Fault\_AsyncParity
    });

// IsExternalAbort()
// =====

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.statuscode) || fault.gpcf.gpf == GPCF\_EABT;
```

## Library pseudocode for shared/functions/aborts/IsExternalSyncAbort

```
// IsExternalSyncAbort()
// =====
// Returns TRUE if the abort currently being processed is an external
// synchronous abort and FALSE otherwise.

boolean IsExternalSyncAbort(Fault statuscode)
    assert statuscode != Fault\_None;

    return (statuscode IN {
        Fault\_SyncExternal,
        Fault\_SyncParity,
        Fault\_SyncExternalOnWalk,
        Fault\_SyncParityOnWalk
    });

// IsExternalSyncAbort()
// =====

boolean IsExternalSyncAbort(FaultRecord fault)
    return IsExternalSyncAbort(fault.statuscode) || fault.gpcf.gpf == GPCF\_EABT;
```

## Library pseudocode for shared/functions/aborts/IsFault

```
// IsFault()
// =====
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.statuscode != Fault_None;

// IsFault()
// =====
// Return TRUE if a fault is associated with a memory access.

boolean IsFault(Fault fault)
    return fault != Fault_None;

// IsFault()
// =====
// Return TRUE if a fault is associated with status returned by memory.

boolean IsFault(PhysMemRetStatus retstatus)
    return retstatus.statuscode != Fault_None;
```

## Library pseudocode for shared/functions/aborts/IsSErrorInterrupt

```
// IsSErrorInterrupt()
// =====
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

boolean IsSErrorInterrupt(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsSErrorInterrupt()
// =====

boolean IsSErrorInterrupt(FaultRecord fault)
    return IsSErrorInterrupt(fault.statuscode);
```

## Library pseudocode for shared/functions/aborts/IsSecondStage

```
// IsSecondStage()
// =====

boolean IsSecondStage(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    return fault.secondstage;
```

## Library pseudocode for shared/functions/aborts/LSInstructionSyndrome

```
// LSInstructionSyndrome()
// =====
// Returns the extended syndrome information for a second stage fault.
// <10> - Syndrome valid bit. The syndrome is valid only for certain types of access instruction.
// <9:8> - Access size.
// <7> - Sign extended (for loads).
// <6:2> - Transfer register.
// <1> - Transfer register is 64-bit.
// <0> - Instruction has acquire/release semantics.

bits(11) LSInstructionSyndrome();
```

## Library pseudocode for shared/functions/aborts/ReportAsGPCException

```
// ReportAsGPCException()
// =====
// Determine whether the given GPCF is reported as a Granule Protection Check Exception
// rather than a Data or Instruction Abort

boolean ReportAsGPCException(FaultRecord fault)
    assert HaveRME();
    assert fault.statuscode IN {Fault\_GPCFOnWalk, Fault\_GPCFOnOutput};
    assert fault.gpcf.gpf != GPCF\_None;

    case fault.gpcf.gpf of
        when GPCF\_Walk           return TRUE;
        when GPCF\_AddressSize return TRUE;
        when GPCF\_EABT          return TRUE;
        when GPCF\_Fail          return SCR_EL3.GPF == '1' && PSTATE.EL != EL3;
```

## Library pseudocode for shared/functions/cache/CACHE\_OP

```
// CACHE_OP()
// =====
// Performs Cache maintenance operations as per CacheRecord.

CACHE_OP(CacheRecord cache)
    IMPLEMENTATION_DEFINED;
```

## Library pseudocode for shared/functions/cache/CPASAtPAS

```
// CPASAtPAS()
// =====
// Get cache PA space for given PA space.

CachePASpace CPASAtPAS(PASpace pas)
    case pas of
        when PAS\_NonSecure
            return CPAS\_NonSecure;
        when PAS\_Secure
            return CPAS\_Secure;
        when PAS\_Root
            return CPAS\_Root;
        when PAS\_Realm
            return CPAS\_Realm;
```

## Library pseudocode for shared/functions/cache/CPASAtSecurityState

```
// CPASAtSecurityState()
// =====
// Get cache PA space for given security state.

CachePASpace CPASAtSecurityState(SecurityState ss)
    case ss of
        when SS\_NonSecure
            return CPAS\_NonSecure;
        when SS\_Secure
            return CPAS\_SecureNonSecure;
        when SS\_Root
            return CPAS\_Any;
        when SS\_Realm
            return CPAS\_RealmNonSecure;
```

## Library pseudocode for shared/functions/cache/CacheRecord

```
// CacheRecord
// =====
// Details related to a cache operation.

type CacheRecord is (
    AccessType      acctype,          // Access type
    CacheOp        cacheop,          // Cache operation
    CacheOpScope    opscope,          // Cache operation type
    CacheType       cachetype,        // Cache type
    bits(64)        regval,
    FullAddress     paddress,
    bits(64)        vaddress,          // For VA operations
    integer         setnum,            // For SW operations
    integer         waynum,            // For SW operations
    integer         level,             // For SW operations
    Shareability    shareability,
    boolean          translated,
    boolean          is_vmid_valid,     // is vmid valid for current context
    bits(16)         vmid,
    boolean          is_asid_valid,     // is asid valid for current context
    bits(16)         asid,
    SecurityState    security,
    // For cache operations to full cache or by setnum/waynum
    // For operations by address, PA space in paddress
    CachePASpace    cpas
)
```

## Library pseudocode for shared/functions/cache/DCInstNeedsTranslation

```
// DCInstNeedsTranslation()
// =====
// Check whether Data Cache operation needs translation.

boolean DCInstNeedsTranslation(CacheOpScope opscope)
    if opscope == CacheOpScope\_PoE then
        return FALSE;

    if opscope == CacheOpScope\_PoPA then
        return FALSE;

    if CLIDR_EL1.LoC == '000' then
        return !(boolean IMPLEMENTATION_DEFINED
            "No fault generated for DC operations if PoC is before any level of cache");

    if CLIDR_EL1.LoUU == '000' && opscope == CacheOpScope\_PoU then
        return !(boolean IMPLEMENTATION_DEFINED
            "No fault generated for DC operations if PoU is before any level of cache");

    return TRUE;
```

## Library pseudocode for shared/functions/cache/DecodeSW

```
// DecodeSW()
// =====
// Decode input value into setnum, waynum and level for SW instructions.

(integer, integer, integer) DecodeSW(bits(64) regval, CacheType cachetype)
    level = UInt(regval[3:1]);
    (setnum, waynum, linesize) = GetCacheInfo(level, cachetype);
    return (setnum, waynum, level);
```

### Library pseudocode for shared/functions/cache/GetCacheInfo

```
// GetCacheInfo()
// =====
// Returns numsets, associativity & linesize.

(integer, integer, integer) GetCacheInfo(integer level, CacheType cachetype);
```

### Library pseudocode for shared/functions/cache/ICInstNeedsTranslation

```
// ICInstNeedsTranslation()
// =====
// Check whether Instruction Cache operation needs translation.

boolean ICInstNeedsTranslation(CacheOpScope opscope)
    return boolean IMPLEMENTATION_DEFINED "Instruction Cache needs translation";
```

### Library pseudocode for shared/functions/common/ASR

```
// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR\_C(x, shift);
    return result;
```

### Library pseudocode for shared/functions/common/ASR\_C

```
// ASR_C()
// =====

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<(shift+N)-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

### Library pseudocode for shared/functions/common/Abs

```
// Abs()
// =====

integer Abs(integer x)
    return if x >= 0 then x else -x;

// Abs()
// =====

real Abs(real x)
    return if x >= 0.0 then x else -x;
```



### Library pseudocode for shared/functions/common/Align

```
// Align()
// =====

integer Align(integer x, integer y)
    return y * (x DIV y);

// Align()
// =====

bits(N) Align(bits(N) x, integer y)
    return Align(UInt(x), y)<N-1:0>;
```

### Library pseudocode for shared/functions/common/BitCount

```
// BitCount()
// =====

integer BitCount(bits(N) x)
    integer result = 0;
    for i = 0 to N-1
        if x<i> == '1' then
            result = result + 1;
    return result;
```

### Library pseudocode for shared/functions/common/CountLeadingSignBits

```
// CountLeadingSignBits()
// =====

integer CountLeadingSignBits(bits(N) x)
    return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);
```

### Library pseudocode for shared/functions/common/CountLeadingZeroBits

```
// CountLeadingZeroBits()
// =====

integer CountLeadingZeroBits(bits(N) x)
    return N - (HighestSetBit(x) + 1);
```

### Library pseudocode for shared/functions/common/Elem

```
// Elem[] - non-assignment form
// =====

bits(size) Elem(bits(N) vector, integer e, integer size)
    assert e >= 0 && (e+1)*size <= N;
    return vector<(e*size+size)-1 : e*size>;

// Elem[] - assignment form
// =====

Elem(bits(N) &vector, integer e, integer size) = bits(size) value
    assert e >= 0 && (e+1)*size <= N;
    vector<(e+1)*size-1:e*size> = value;
    return;
```

### Library pseudocode for shared/functions/common/Extend

```
// Extend()
// =====

bits(N) Extend(bits(M) x, integer N, boolean unsigned)
    return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);
```

### Library pseudocode for shared/functions/common/HighestSetBit

```
// HighestSetBit()
// =====

integer HighestSetBit(bits(N) x)
  for i = N-1 downto 0
    if x<i> == '1' then return i;
  return -1;
```

### Library pseudocode for shared/functions/common/Int

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
  result = if unsigned then UInt(x) else SInt(x);
  return result;
```

### Library pseudocode for shared/functions/common/IsAligned

```
// IsAligned()
// =====

boolean IsAligned(bits(N) x, integer y)
  return x == Align(x, y);
```

### Library pseudocode for shared/functions/common/IsOnes

```
// IsOnes()
// =====

boolean IsOnes(bits(N) x)
  return x == Ones(N);
```

### Library pseudocode for shared/functions/common/IsZero

```
// IsZero()
// =====

boolean IsZero(bits(N) x)
  return x == Zeros(N);
```

### Library pseudocode for shared/functions/common/IsZeroBit

```
// IsZeroBit()
// =====

bit IsZeroBit(bits(N) x)
  return if IsZero(x) then '1' else '0';
```

### Library pseudocode for shared/functions/common/LSL

```
// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
  assert shift >= 0;
  bits(N) result;
  if shift == 0 then
    result = x;
  else
    (result, -) = LSL\_C(x, shift);
  return result;
```

### Library pseudocode for shared/functions/common/LSL\_C

```
// LSL_C()
// =====

(bits(N), bit) LSL_C(bits(N) x, integer shift)
  assert shift > 0 && shift < 256;
  extended_x = x : Zeros(shift);
  result = extended_x<N-1:0>;
  carry_out = extended_x<N>;
  return (result, carry_out);
```

### Library pseudocode for shared/functions/common/LSR

```
// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
  assert shift >= 0;
  bits(N) result;
  if shift == 0 then
    result = x;
  else
    (result, -) = LSR\_C(x, shift);
  return result;
```

### Library pseudocode for shared/functions/common/LSR\_C

```
// LSR_C()
// =====

(bits(N), bit) LSR_C(bits(N) x, integer shift)
  assert shift > 0 && shift < 256;
  extended_x = ZeroExtend(x, shift+N);
  result = extended_x<(shift+N)-1:shift>;
  carry_out = extended_x<shift-1>;
  return (result, carry_out);
```

### Library pseudocode for shared/functions/common/LowestSetBit

```
// LowestSetBit()
// =====

integer LowestSetBit(bits(N) x)
  for i = 0 to N-1
    if x<i> == '1' then return i;
  return N;
```

### Library pseudocode for shared/functions/common/Max

```
// Max()
// =====

integer Max(integer a, integer b)
  return if a >= b then a else b;

// Max()
// =====

real Max(real a, real b)
  return if a >= b then a else b;
```

### Library pseudocode for shared/functions/common/Min

```
// Min()
// =====

integer Min(integer a, integer b)
    return if a <= b then a else b;

// Min()
// =====

real Min(real a, real b)
    return if a <= b then a else b;
```

### Library pseudocode for shared/functions/common/Ones

```
// Ones()
// =====

bits(N) Ones(integer N)
    return Replicate('1',N);
```

### Library pseudocode for shared/functions/common/ROR

```
// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = ROR\_C(x, shift);
    return result;
```

### Library pseudocode for shared/functions/common/ROR\_C

```
// ROR_C()
// =====

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0 && shift < 256;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```

### Library pseudocode for shared/functions/common/RShr

```
// RShr()
// =====
// Shift integer value right with rounding

integer RShr(integer value, integer shift, boolean round)
    assert shift > 0;
    if round then
        return (value + (1 << (shift - 1))) >> shift;
    else
        return value >> shift;
```

### Library pseudocode for shared/functions/common/Replicate

```
// Replicate()
// =====

bits(M*N) Replicate(bits(M) x, integer N);
```

### Library pseudocode for shared/functions/common/RoundDown

```
// RoundDown()
// =====

integer RoundDown(real x);
```

### Library pseudocode for shared/functions/common/RoundTowardsZero

```
// RoundTowardsZero()
// =====

integer RoundTowardsZero(real x)
  return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);
```

### Library pseudocode for shared/functions/common/RoundUp

```
// RoundUp()
// =====

integer RoundUp(real x);
```

### Library pseudocode for shared/functions/common/SInt

```
// SInt()
// =====

integer SInt(bits(N) x)
  result = 0;
  for i = 0 to N-1
    if x<i> == '1' then result = result + 2^i;
  if x<N-1> == '1' then result = result - 2^N;
  return result;
```

### Library pseudocode for shared/functions/common/SignExtend

```
// SignExtend()
// =====

bits(N) SignExtend(bits(M) x, integer N)
  assert N >= M;
  return Replicate(x<M-1>, N-M) : x;
```

### Library pseudocode for shared/functions/common/Signal

```
// Signal
// =====
// Available signal types

enumeration Signal {Signal_Low, Signal_High};
```

### Library pseudocode for shared/functions/common/Split

```
// Split()
// =====

(bits(M-N), bits(N)) Split(bits(M) value, integer N)
    assert M > N;
    return (value<M-1:N>, value<N-1:0>);
```

### Library pseudocode for shared/functions/common/UInt

```
// UInt()
// =====

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

### Library pseudocode for shared/functions/common/ZeroExtend

```
// ZeroExtend()
// =====

bits(N) ZeroExtend(bits(M) x, integer N)
    assert N >= M;
    return Zeros(N-M) : x;
```

### Library pseudocode for shared/functions/common/Zeros

```
// Zeros()
// =====

bits(N) Zeros(integer N)
    return Replicate('0', N);
```

## Library pseudocode for shared/functions/counters/AArch32.CheckTimerConditions

```
// AArch32.CheckTimerConditions()
// =====
// Checking timer conditions for all A32 timer registers

AArch32.CheckTimerConditions()
    boolean status;
    bits(64) offset;
    offset = Zeros(64);
    assert !HaveAArch64();

    if HaveEL(EL3) then
        if CNTP_CTL_S.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL_S,
                                           CNTP_CTL_S.IMASK, InterruptID_CNTPS);
            CNTP_CTL_S.ISTATUS = if status then '1' else '0';

        if CNTP_CTL_NS.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL_NS,
                                           CNTP_CTL_NS.IMASK, InterruptID_CNTP);
            CNTP_CTL_NS.ISTATUS = if status then '1' else '0';
    else
        if CNTP_CTL.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL,
                                           CNTP_CTL.IMASK, InterruptID_CNTP);
            CNTP_CTL.ISTATUS = if status then '1' else '0';

    if HaveEL(EL2) && CNTHP_CTL.ENABLE == '1' then
        status = IsTimerConditionMet(offset, CNTHP_CVAL,
                                       CNTHP_CTL.IMASK, InterruptID_CNTHP);
        CNTHP_CTL.ISTATUS = if status then '1' else '0';

    if CNTV_CTL_EL0.ENABLE == '1' then
        status = IsTimerConditionMet(CNTVOFF_EL2, CNTV_CVAL_EL0,
                                       CNTV_CTL_EL0.IMASK, InterruptID_CNTV);
        CNTV_CTL_EL0.ISTATUS = if status then '1' else '0';

    return;
```

## Library pseudocode for shared/functions/counters/AArch64.CheckTimerConditions

```
// AArch64.CheckTimerConditions()
// =====
// Checking timer conditions for all A64 timer registers

AArch64.CheckTimerConditions()
    boolean status;
    bits(64) offset;
    bit imask;
    SecurityState ss = CurrentSecurityState();
    boolean ecv = FALSE;
    if HaveECVExt() then
        ecv = CNTHCTL_EL2.ECV == '1' && SCR_EL3.ECVEn == '1' && EL2Enabled();
    if ecv then
        offset = CNTPOFF_EL2;
    else
        offset = Zeros(64);
    if CNTP_CTL_EL0.ENABLE == '1' then
        imask = CNTP_CTL_EL0.IMASK;
        if HaveRME() && ss IN {SS\_Root, SS\_Realm} && CNTHCTL_EL2.CNTPMASK == '1' then
            imask = '1';
        status = IsTimerConditionMet(offset, CNTP_CVAL_EL0,
                                     imask, InterruptID\_CNTP);
        CNTP_CTL_EL0.ISTATUS = if status then '1' else '0';
    if ((HaveEL(EL3) || (HaveEL(EL2) && !HaveSecureEL2Ext())) &&
        CNTHP_CTL_EL2.ENABLE == '1') then
        status = IsTimerConditionMet(Zeros(64), CNTHP_CVAL_EL2,
                                     CNTHP_CTL_EL2.IMASK, InterruptID\_CNTHP);
        CNTHP_CTL_EL2.ISTATUS = if status then '1' else '0';
    if HaveEL(EL2) && HaveSecureEL2Ext() && CNTHPS_CTL_EL2.ENABLE == '1' then
        status = IsTimerConditionMet(Zeros(64), CNTHPS_CVAL_EL2,
                                     CNTHPS_CTL_EL2.IMASK, InterruptID\_CNTHPS);
        CNTHPS_CTL_EL2.ISTATUS = if status then '1' else '0';

    if CNTPS_CTL_EL1.ENABLE == '1' then
        status = IsTimerConditionMet(offset, CNTPS_CVAL_EL1,
                                     CNTPS_CTL_EL1.IMASK, InterruptID\_CNTPS);
        CNTPS_CTL_EL1.ISTATUS = if status then '1' else '0';

    if CNTV_CTL_EL0.ENABLE == '1' then
        imask = CNTV_CTL_EL0.IMASK;
        if HaveRME() && ss IN {SS\_Root, SS\_Realm} && CNTHCTL_EL2.CNTVMASK == '1' then
            imask = '1';
        status = IsTimerConditionMet(CNTVOFF_EL2, CNTV_CVAL_EL0,
                                     imask, InterruptID\_CNTV);
        CNTV_CTL_EL0.ISTATUS = if status then '1' else '0';

    if ((HaveVirtHostExt() && (HaveEL(EL3) || !HaveSecureEL2Ext())) &&
        CNTHV_CTL_EL2.ENABLE == '1') then
        status = IsTimerConditionMet(Zeros(64), CNTHV_CVAL_EL2,
                                     CNTHV_CTL_EL2.IMASK, InterruptID\_CNTHV);
        CNTHV_CTL_EL2.ISTATUS = if status then '1' else '0';

    if ((HaveSecureEL2Ext() && HaveVirtHostExt()) &&
        CNTHVS_CTL_EL2.ENABLE == '1') then
        status = IsTimerConditionMet(Zeros(64), CNTHVS_CVAL_EL2,
                                     CNTHVS_CTL_EL2.IMASK, InterruptID\_CNTHVS);
        CNTHVS_CTL_EL2.ISTATUS = if status then '1' else '0';
    return;
```



## Library pseudocode for shared/functions/counters/GenericCounterTick

```
// GenericCounterTick()
// =====
// Increments PhysicalCount value for every clock tick.

GenericCounterTick()
    bits(64) prev_physical_count;
    if CNTCR.EN == '0' then
        if !HaveAArch64() then
            AArch32.CheckTimerConditions();
        else
            AArch64.CheckTimerConditions();
        return;
    prev_physical_count = PhysicalCountInt();
    if HaveCNTSCEExt() && CNTCR.SCEN == '1' then
        PhysicalCount = PhysicalCount + ZeroExtend(CNTSCR, 88);
    else
        PhysicalCount<87:24> = PhysicalCount<87:24> + 1;
    if !HaveAArch64() then
        AArch32.CheckTimerConditions();
    else
        AArch64.CheckTimerConditions();
    TestEventCNTP(prev_physical_count, PhysicalCountInt());
    TestEventCNTV(prev_physical_count, PhysicalCountInt());
    return;
```

## Library pseudocode for shared/functions/counters/IsTimerConditionMet

```
// IsTimerConditionMet()
// =====

boolean IsTimerConditionMet(bits(64) offset, bits(64) compare_value,
                           bits(1) imask, InterruptID intid)
    boolean condition_met;
    Signal level;
    condition_met = (UInt(PhysicalCountInt() - offset) -
                    UInt(compare_value)) >= 0;
    level = if condition_met && imask == '0' then Signal_High else Signal_Low;
    SetInterruptRequestLevel(intid, level);
    return condition_met;
```

## Library pseudocode for shared/functions/counters/PhysicalCount

```
bits(88) PhysicalCount;
```

## Library pseudocode for shared/functions/counters/SetEventRegister

```
// SetEventRegister()
// =====
// Sets the Event Register of this PE

SetEventRegister()
    EventRegister = '1';
    return;
```

## Library pseudocode for shared/functions/counters/TestEventCNTP

```
// TestEventCNTP()
// =====
// Generate Event stream from the physical counter

TestEventCNTP(bits(64) prev_physical_count, bits(64) current_physical_count)
  bits(64) offset;
  bits(1) samplebit, previousbit;
  if CNTHCTL_EL2.EVNTEN == '1' then
    n = UInt(CNTHCTL_EL2.EVNTI);
    if HaveECVExt() && CNTHCTL_EL2.EVNTIS == '1' then
      n = n + 8;
    boolean ecv = FALSE;
    if HaveECVExt() then
      ecv = (EL2Enabled() && CNTHCTL_EL2.ECV == '1' &&
        SCR_EL3.ECVEn == '1');
      offset = if ecv then CNTPOFF_EL2 else Zeros(64);
    samplebit = (current_physical_count - offset)<n>;
    previousbit = (prev_physical_count - offset)<n>;
    if CNTHCTL_EL2.EVNTDIR == '0' then
      if previousbit == '0' && samplebit == '1' then SetEventRegister();
    else
      if previousbit == '1' && samplebit == '0' then SetEventRegister();
  return;
```

## Library pseudocode for shared/functions/counters/TestEventCNTV

```
// TestEventCNTV()
// =====
// Generate Event stream from the virtual counter

TestEventCNTV(bits(64) prev_physical_count, bits(64) current_physical_count)
  bits(64) offset;
  bits(1) samplebit, previousbit;
  if (!(HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11') &&
    CNTKCTL_EL1.EVNTEN == '1') then
    n = UInt(CNTKCTL_EL1.EVNTI);
    if HaveECVExt() && CNTKCTL_EL1.EVNTIS == '1' then
      n = n + 8;
    if HaveEL(EL2) && (!EL2Enabled() || HCR_EL2.<E2H,TGE> != '11') then
      offset = CNTVOFF_EL2;
    else
      offset = Zeros(64);
    samplebit = (current_physical_count - offset)<n>;
    previousbit = (prev_physical_count - offset)<n>;
    if CNTKCTL_EL1.EVNTDIR == '0' then
      if previousbit == '0' && samplebit == '1' then SetEventRegister();
    else
      if previousbit == '1' && samplebit == '0' then SetEventRegister();
  return;
```

## Library pseudocode for shared/functions/crc/BitReverse

```
// BitReverse()
// =====

bits(N) BitReverse(bits(N) data)
  bits(N) result;
  for i = 0 to N-1
    result<(N-i)-1> = data<i>;
  return result;
```

## Library pseudocode for shared/functions/crc/HaveCRCExt

```
// HaveCRCExt()
// =====

boolean HaveCRCExt()
    return IsFeatureImplemented(FEAT_CRC32);
```

## Library pseudocode for shared/functions/crc/Poly32Mod2

```
// Poly32Mod2()
// =====

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

bits(32) Poly32Mod2(bits(N) data_in, bits(32) poly)
    assert N > 32;
    bits(N) data = data_in;
    for i = N-1 downto 32
        if data<i> == '1' then
            data<i-1:0> = data<i-1:0> EOR (poly:Zeros(i-32));
    return data<31:0>;
```

## Library pseudocode for shared/functions/crypto/AESInvMixColumns

```
// AESInvMixColumns()
// =====
// Transformation in the Inverse Cipher that is the inverse of AESMixColumns.

bits(128) AESInvMixColumns(bits (128) op)
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> = (FFmul0E(in0<c*8+:8>) EOR FFmul0B(in1<c*8+:8>) EOR FFmul0D(in2<c*8+:8>) EOR
            FFmul09(in3<c*8+:8>));
        out1<c*8+:8> = (FFmul09(in0<c*8+:8>) EOR FFmul0E(in1<c*8+:8>) EOR FFmul0B(in2<c*8+:8>) EOR
            FFmul0D(in3<c*8+:8>));
        out2<c*8+:8> = (FFmul0D(in0<c*8+:8>) EOR FFmul09(in1<c*8+:8>) EOR FFmul0E(in2<c*8+:8>) EOR
            FFmul0B(in3<c*8+:8>));
        out3<c*8+:8> = (FFmul0B(in0<c*8+:8>) EOR FFmul0D(in1<c*8+:8>) EOR FFmul09(in2<c*8+:8>) EOR
            FFmul0E(in3<c*8+:8>));

    return (
        out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
        out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
        out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
    );
```

## Library pseudocode for shared/functions/crypto/AESInvShiftRows

```
// AESInvShiftRows()
// =====
// Transformation in the Inverse Cipher that is inverse of AESShiftRows.

bits(128) AESInvShiftRows(bits(128) op)
    return (
        op< 31: 24> : op< 55: 48> : op< 79: 72> : op<103: 96> :
        op<127:120> : op< 23: 16> : op< 47: 40> : op< 71: 64> :
        op< 95: 88> : op<119:112> : op< 15: 8> : op< 39: 32> :
        op< 63: 56> : op< 87: 80> : op<111:104> : op< 7: 0>
    );
```

## Library pseudocode for shared/functions/crypto/AESInvSubBytes

```
// AESInvSubBytes()
// =====
// Transformation in the Inverse Cipher that is the inverse of AESSubBytes.

bits(128) AESInvSubBytes(bits(128) op)
    // Inverse S-box values
    bits(16*16*8) GF2_inv = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x7d0c2155631469e126d677ba7e042b17<127:0> :
        /*E*/ 0x619953833cbbec8b0f52aae4d3be0a0<127:0> :
        /*D*/ 0xef9cc9939f7ae52d0d4ab519a97f5160<127:0> :
        /*C*/ 0x5fec8027591012b131c7078833a8dd1f<127:0> :
        /*B*/ 0xf45acd78fec0db9a2079d2c64b3e56fc<127:0> :
        /*A*/ 0x1bbe18aa0e62b76f89c5291d711af147<127:0> :
        /*9*/ 0x6edf751ce837f9e28535ade72274ac96<127:0> :
        /*8*/ 0x73e6b4f0cecff297eadc674f4111913a<127:0> :
        /*7*/ 0x6b8a130103bdafc1020f3fca8f1e2cd0<127:0> :
        /*6*/ 0x0645b3b80558e4f70ad3bc8c00abd890<127:0> :
        /*5*/ 0x849d8da75746155edab9edfd5048706c<127:0> :
        /*4*/ 0x92b6655dcc5ca4d41698688664f6f872<127:0> :
        /*3*/ 0x25d18b6d49a25b76b224d92866a12e08<127:0> :
        /*2*/ 0x4ec3fa420b954cee3d23c2a632947b54<127:0> :
        /*1*/ 0xcbe9dec444438e3487ff2f9b8239e37c<127:0> :
        /*0*/ 0xfbd7f3819ea340bf38a53630d56a0952<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2_inv<UInt>(op<i*8+:8>)*8+:8>;
    return out;
```

## Library pseudocode for shared/functions/crypto/AESMixColumns

```
// AESMixColumns()
// =====
// Transformation in the Cipher that takes all of the columns of the
// State and mixes their data (independently of one another) to
// produce new columns.

bits(128) AESMixColumns(bits (128) op)
  bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
  bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
  bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
  bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

  bits(4*8) out0;
  bits(4*8) out1;
  bits(4*8) out2;
  bits(4*8) out3;

  for c = 0 to 3
    out0<c*8+:8> = (FFmul02(in0<c*8+:8>) EOR FFmul03(in1<c*8+:8>) EOR
                  in2<c*8+:8> EOR in3<c*8+:8>);
    out1<c*8+:8> = (FFmul02(in1<c*8+:8>) EOR FFmul03(in2<c*8+:8>) EOR
                  in3<c*8+:8> EOR in0<c*8+:8>);
    out2<c*8+:8> = (FFmul02(in2<c*8+:8>) EOR FFmul03(in3<c*8+:8>) EOR
                  in0<c*8+:8> EOR in1<c*8+:8>);
    out3<c*8+:8> = (FFmul02(in3<c*8+:8>) EOR FFmul03(in0<c*8+:8>) EOR
                  in1<c*8+:8> EOR in2<c*8+:8>);

  return (
    out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
    out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
    out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
    out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
  );
```

## Library pseudocode for shared/functions/crypto/AESShiftRows

```
// AESShiftRows()
// =====
// Transformation in the Cipher that processes the State by cyclically
// shifting the last three rows of the State by different offsets.

bits(128) AESShiftRows(bits(128) op)
  return (
    op< 95: 88> : op< 55: 48> : op< 15:  8> : op<103: 96> :
    op< 63: 56> : op< 23: 16> : op<111:104> : op< 71: 64> :
    op< 31: 24> : op<119:112> : op< 79: 72> : op< 39: 32> :
    op<127:120> : op< 87: 80> : op< 47: 40> : op<  7:  0>
  );
```

## Library pseudocode for shared/functions/crypto/AESSubBytes

```
// AESSubBytes()
// =====
// Transformation in the Cipher that processes the State using a nonlinear
// byte substitution table (S-box) that operates on each of the State bytes
// independently.

bits(128) AESSubBytes(bits(128) op)
    // S-box values
    bits(16*16*8) GF2 = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x16bb54b00f2d99416842e6bf0d89a18c<127:0> :
        /*E*/ 0xdf2855cee9871e9b948ed9691198f8e1<127:0> :
        /*D*/ 0x9e1dc186b95735610ef6034866b53e70<127:0> :
        /*C*/ 0x8a8bbd4b1f74dde8c6b4a61c2e2578ba<127:0> :
        /*B*/ 0x08ae7a65eaf4566ca94ed58d6d37c8e7<127:0> :
        /*A*/ 0x79e4959162acd3c25c2406490a3a32e0<127:0> :
        /*9*/ 0xdb0b5ede14b8ee4688902a22dc4f8160<127:0> :
        /*8*/ 0x73195d643d7ea7c41744975fec130ccd<127:0> :
        /*7*/ 0xd2f3ff1021dab6bcf5389d928f40a351<127:0> :
        /*6*/ 0xa89f3c507f02f94585334d43fbaaefd0<127:0> :
        /*5*/ 0xcf584c4a39becb6a5bb1fc20ed00d153<127:0> :
        /*4*/ 0x842fe329b3d63b52a05a6e1b1a2c8309<127:0> :
        /*3*/ 0x75b227ebe28012079a059618c323c704<127:0> :
        /*2*/ 0x1531d871f1e5a534ccf73f362693fdb7<127:0> :
        /*1*/ 0xc072a49cafa2d4adf04759fa7dc982ca<127:0> :
        /*0*/ 0x76abd7fe2b670130c56f6bf27b777c63<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2<UInt>(op<i*8+:8>)*8+:8>;
    return out;
```

## Library pseudocode for shared/functions/crypto/FFmul02

```
// FFmul02()
// =====

bits(8) FFmul02(bits(8) b)
    bits(256*8) FFmul_02 = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0xE5E7E1E3EDEFE9EBF5F7F1F3FDFFF9FB<127:0> :
        /*E*/ 0xC5C7C1C3CDCFC9CBD5D7D1D3DDDFD9DB<127:0> :
        /*D*/ 0xA5A7A1A3ADAF9ABB5B7B1B3BDBFB9BB<127:0> :
        /*C*/ 0x858781838D8F898B959791939D9F999B<127:0> :
        /*B*/ 0x656761636D6F696B757771737D7F797B<127:0> :
        /*A*/ 0x454741434D4F494B555751535D5F595B<127:0> :
        /*9*/ 0x252721232D2F292B353731333D3F393B<127:0> :
        /*8*/ 0x050701030D0F090B151711131D1F191B<127:0> :
        /*7*/ 0xFEFCFAF8F6F4F2F0EEECFAE8E6E4E2E0<127:0> :
        /*6*/ 0xDEDCDAD8D6D4D2D0CECCAC8C6C4C2C0<127:0> :
        /*5*/ 0xBEBBCBAB8B6B4B2B0AEACAAA8A6A4A2A0<127:0> :
        /*4*/ 0x9E9C9A98969492908E8C8A8886848280<127:0> :
        /*3*/ 0x7E7C7A78767472706E6C6A6866646260<127:0> :
        /*2*/ 0x5E5C5A58565452504E4C4A4846444240<127:0> :
        /*1*/ 0x3E3C3A38363432302E2C2A2826242220<127:0> :
        /*0*/ 0x1E1C1A1816141210E0C0A0806040200<127:0>
    );
    return FFmul_02<UInt>(b)*8+:8>;
```

## Library pseudocode for shared/functions/crypto/FFmul03

```
// FFmul03()
// =====

bits(8) FFmul03(bits(8) b)
    bits(256*8) FFmul_03 = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x1A191C1F16151013020104070E0D080B<127:0> :
        /*E*/ 0x2A292C2F26252023323134373E3D383B<127:0> :
        /*D*/ 0x7A797C7F76757073626164676E6D686B<127:0> :
        /*C*/ 0x4A494C4F46454043525154575E5D585B<127:0> :
        /*B*/ 0xDAD9DCDFD6D5D0D3C2C1C4C7CECDC8CB<127:0> :
        /*A*/ 0xEAE9ECEFE6E5E0E3F2F1F4F7FEFDF8FB<127:0> :
        /*9*/ 0xBAB9BCBFB6B5B0B3A2A1A4A7AEADA8AB<127:0> :
        /*8*/ 0x8A898C8F86858083929194979E9D989B<127:0> :
        /*7*/ 0x818287848D8E8B88999A9F9C95969390<127:0> :
        /*6*/ 0xB1B2B7B4DBEBBBB8A9AAAFACA5A6A3A0<127:0> :
        /*5*/ 0xE1E2E7E4EDEEEBE8F9FAFFFCCF5F6F3F0<127:0> :
        /*4*/ 0xD1D2D7D4DDDEDBD8C9CACFCCC5C6C3C0<127:0> :
        /*3*/ 0x414247444D4E4B48595A5F5C55565350<127:0> :
        /*2*/ 0x717277747D7E7B78696A6F6C65666360<127:0> :
        /*1*/ 0x212227242D2E2B28393A3F3C35363330<127:0> :
        /*0*/ 0x111217141D1E1B18090A0F0C05060300<127:0>
    );
    return FFmul_03<UInt>(b)*8+:8>;
```

## Library pseudocode for shared/functions/crypto/FFmul09

```
// FFmul09()
// =====

bits(8) FFmul09(bits(8) b)
    bits(256*8) FFmul_09 = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x464F545D626B70790E071C152A233831<127:0> :
        /*E*/ 0xD6DFC4CDF2FBE0E99E978C85BAB3A8A1<127:0> :
        /*D*/ 0x7D746F6659504B42353C272E1118030A<127:0> :
        /*C*/ 0xEDE4FFF6C9C0DBD2A5ACB7BE8188939A<127:0> :
        /*B*/ 0x3039222B141D060F78716A635C554E47<127:0> :
        /*A*/ 0xA0A9B2BB848D969FE8E1FAF3CCC5DED7<127:0> :
        /*9*/ 0x0B0219102F263D34434A5158676E757C<127:0> :
        /*8*/ 0x9B928980BFB6ADA4D3DAC1C8F7FEE5EC<127:0> :
        /*7*/ 0xAAA3B8B18E879C95E2EBF0F9C6CFD4DD<127:0> :
        /*6*/ 0x3A3328211E170C05727B6069565F444D<127:0> :
        /*5*/ 0x9198838AB5BCA7AED9D0CBC2FDF4EFE6<127:0> :
        /*4*/ 0x0108131A252C373E49405B526D647F76<127:0> :
        /*3*/ 0xDCD5CEC7F8F1EAE3949D868FB0B9A2AB<127:0> :
        /*2*/ 0x4C455E5768617A73040D161F2029323B<127:0> :
        /*1*/ 0xE7EEF5FCC3CAD1D8AFA6BDB48B829990<127:0> :
        /*0*/ 0x777E656C535A41483F362D241B120900<127:0>
    );
    return FFmul_09<UInt>(b)*8+:8>;
```

## Library pseudocode for shared/functions/crypto/FFmul0B

```
// FFmul0B()
// =====

bits(8) FFmul0B(bits(8) b)
    bits(256*8) FFmul_0B = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0xA3A8B5BE8F849992FBF0EDE6D7DCC1CA<127:0> :
        /*E*/ 0x1318050E3F3429224B405D56676C717A<127:0> :
        /*D*/ 0xD8D3CEC5F4FFE2E9808B969DACA7BAB1<127:0> :
        /*C*/ 0x68637E75444F5259303B262D1C170A01<127:0> :
        /*B*/ 0x555E434879726F640D061B10212A373C<127:0> :
        /*A*/ 0xE5EEF3F8C9C2DFD4BDB6ABA0919A878C<127:0> :
        /*9*/ 0x2E2538330209141F767D606B5A514C47<127:0> :
        /*8*/ 0x9E958883B2B9A4AFC6CDD0DBEAE1FCF7<127:0> :
        /*7*/ 0x545F424978736E650C071A11202B363D<127:0> :
        /*6*/ 0xE4EFF2F9C8C3DED5BCB7AAA1909B868D<127:0> :
        /*5*/ 0x2F2439320308151E777C616A5B504D46<127:0> :
        /*4*/ 0x9F948982B3B8A5AEC7CCD1DAEBE0FDF6<127:0> :
        /*3*/ 0xA2A9B4BF8E859893FAF1ECE7D6DDC0CB<127:0> :
        /*2*/ 0x1219040F3E3528234A415C57666D707B<127:0> :
        /*1*/ 0xD9D2CFC4F5FEE3E8818A979CADA6BBB0<127:0> :
        /*0*/ 0x69627F74454E5358313A272C1D160B00<127:0>

    );
    return FFmul_0B<UInt>(b)*8+:8;
```

## Library pseudocode for shared/functions/crypto/FFmul0D

```
// FFmul0D()
// =====

bits(8) FFmul0D(bits(8) b)
    bits(256*8) FFmul_0D = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x979A8D80A3AEB9B4FFF2E5E8CBC6D1DC<127:0> :
        /*E*/ 0x474A5D50737E69642F2235381B16010C<127:0> :
        /*D*/ 0x2C21363B1815020F44495E53707D6A67<127:0> :
        /*C*/ 0xFCF1E6EBC8C5D2DF94998E83A0ADBAB7<127:0> :
        /*B*/ 0xFAF7E0EDCEC3D4D9929F8885A6ABBCB1<127:0> :
        /*A*/ 0x2A27303D1E130409424F5855767B6C61<127:0> :
        /*9*/ 0x414C5B5675786F622924333E1D10070A<127:0> :
        /*8*/ 0x919C8B86A5A8BFB2F9F4E3EECDC0D7DA<127:0> :
        /*7*/ 0x4D40575A7974636E25283F32111C0B06<127:0> :
        /*6*/ 0x9D90878AA9A4B3BEF5F8EFE2C1CCDBD6<127:0> :
        /*5*/ 0xF6FBECE1C2CFD8D59E938489AAA7B0BD<127:0> :
        /*4*/ 0x262B3C31121F08054E4354597A77606D<127:0> :
        /*3*/ 0x202D3A3714190E034845525F7C71666B<127:0> :
        /*2*/ 0xF0FDEAE7C4C9DED39895828FACA1B6BB<127:0> :
        /*1*/ 0x9B96818CAFA2B5B8F3FEE9E4C7CADD0<127:0> :
        /*0*/ 0x4B46515C7F726568232E3934171A0D00<127:0>

    );
    return FFmul_0D<UInt>(b)*8+:8;
```



## Library pseudocode for shared/functions/crypto/FFmul0E

```
// FFmul0E()
// =====

bits(8) FFmul0E(bits(8) b)
    bits(256*8) FFmul_0E = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x8D83919FB5BBA9A7FDF3E1EFC5CBD9D7<127:0> :
        /*E*/ 0x6D63717F555B49471D13010F252B3937<127:0> :
        /*D*/ 0x56584A446E60727C26283A341E10020C<127:0> :
        /*C*/ 0xB6B8AAA48E80929CC6C8DAD4FEF0E2EC<127:0> :
        /*B*/ 0x202E3C321816040A505E4C426866747A<127:0> :
        /*A*/ 0xC0CEDCD2F8F6E4EAB0BEACA28886949A<127:0> :
        /*9*/ 0xFBF5E7E9C3CDDFD18B859799B3BDFAFA1<127:0> :
        /*8*/ 0x1B150709232D3F316B657779535D4F41<127:0> :
        /*7*/ 0xCCC2D0DEF4FAE8E6BCB2A0AE848A9896<127:0> :
        /*6*/ 0x2C22303E141A08065C52404E646A7876<127:0> :
        /*5*/ 0x17190B052F21333D67697B755F51434D<127:0> :
        /*4*/ 0xF7F9EBE5CFC1D3DD87899B95BFB1A3AD<127:0> :
        /*3*/ 0x616F7D735957454B111F0D032927353B<127:0> :
        /*2*/ 0x818F9D93B9B7A5ABF1FFEDE3C9C7D5DB<127:0> :
        /*1*/ 0xBAB4A6A8828C9E90CAC4D6D8F2FCEEE0<127:0> :
        /*0*/ 0x5A544648626C7E702A243638121C0E00<127:0>
    );
    return FFmul_0E<UInt>(b)*8+:8>;
```

## Library pseudocode for shared/functions/crypto/HaveAESExt

```
// HaveAESExt()
// =====
// TRUE if AES cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveAESExt()
    return IsFeatureImplemented(FEAT_AES);
```

## Library pseudocode for shared/functions/crypto/HaveBit128PMULLExt

```
// HaveBit128PMULLExt()
// =====
// TRUE if 128 bit form of PMULL instructions support is implemented,
// FALSE otherwise.

boolean HaveBit128PMULLExt()
    return IsFeatureImplemented(FEAT_PMULL);
```

## Library pseudocode for shared/functions/crypto/HaveSHA1Ext

```
// HaveSHA1Ext()
// =====
// TRUE if SHA1 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA1Ext()
    return IsFeatureImplemented(FEAT_SHA1);
```

## Library pseudocode for shared/functions/crypto/HaveSHA256Ext

```
// HaveSHA256Ext()
// =====
// TRUE if SHA256 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA256Ext()
    return IsFeatureImplemented(FEAT_SHA256);
```

### Library pseudocode for shared/functions/crypto/HaveSHA3Ext

```
// HaveSHA3Ext()
// =====
// TRUE if SHA3 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA3Ext()
    return IsFeatureImplemented(FEAT_SHA3);
```

### Library pseudocode for shared/functions/crypto/HaveSHA512Ext

```
// HaveSHA512Ext()
// =====
// TRUE if SHA512 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA512Ext()
    return IsFeatureImplemented(FEAT_SHA512);
```

### Library pseudocode for shared/functions/crypto/HaveSM3Ext

```
// HaveSM3Ext()
// =====
// TRUE if SM3 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM3Ext()
    return IsFeatureImplemented(FEAT_SM3);
```

### Library pseudocode for shared/functions/crypto/HaveSM4Ext

```
// HaveSM4Ext()
// =====
// TRUE if SM4 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM4Ext()
    return IsFeatureImplemented(FEAT_SM4);
```

### Library pseudocode for shared/functions/crypto/ROL

```
// ROL()
// =====

bits(N) ROL(bits(N) x, integer shift)
    assert shift >= 0 && shift <= N;
    if (shift == 0) then
        return x;
    return ROR(x, N-shift);
```

### Library pseudocode for shared/functions/crypto/SHA256hash

```
// SHA256hash()
// =====

bits(128) SHA256hash(bits (128) x_in, bits(128) y_in, bits(128) w, boolean part1)
  bits(32) chs, maj, t;
  bits(128) x = x_in;
  bits(128) y = y_in;

  for e = 0 to 3
    chs = SHAchoose(y<31:0>, y<63:32>, y<95:64>);
    maj = SHAmajority(x<31:0>, x<63:32>, x<95:64>);
    t = y<127:96> + SHAhashSIGMA1(y<31:0>) + chs + Elem[w, e, 32];
    x<127:96> = t + x<127:96>;
    y<127:96> = t + SHAhashSIGMA0(x<31:0>) + maj;
    <y, x> = ROL(y : x, 32);
  return (if part1 then x else y);
```

### Library pseudocode for shared/functions/crypto/SHAchoose

```
// SHAchoose()
// =====

bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
  return ((y EOR z) AND x) EOR z;
```

### Library pseudocode for shared/functions/crypto/SHAhashSIGMA0

```
// SHAhashSIGMA0()
// =====

bits(32) SHAhashSIGMA0(bits(32) x)
  return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);
```

### Library pseudocode for shared/functions/crypto/SHAhashSIGMA1

```
// SHAhashSIGMA1()
// =====

bits(32) SHAhashSIGMA1(bits(32) x)
  return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);
```

### Library pseudocode for shared/functions/crypto/SHAmajority

```
// SHAmajority()
// =====

bits(32) SHAmajority(bits(32) x, bits(32) y, bits(32) z)
  return ((x AND y) OR ((x OR y) AND z));
```

### Library pseudocode for shared/functions/crypto/SHAparity

```
// SHAparity()
// =====

bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
  return (x EOR y EOR z);
```

## Library pseudocode for shared/functions/crypto/Sbox

```
// Sbox()
// =====
// Used in SM4E crypto instruction

bits(8) Sbox(bits(8) sboxin)
    bits(8) sboxout;
    bits(2048) sboxstring = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0xd690e9fecce13db716b614c228fb2c05<127:0> :
        /*E*/ 0x2b679a762abe04c3aa44132649860699<127:0> :
        /*D*/ 0x9c4250f491ef987a33540b43edcfac62<127:0> :
        /*C*/ 0xe4b31ca9c908e89580df94fa758f3fa6<127:0> :
        /*B*/ 0x4707a7fcf37317ba83593c19e6854fa8<127:0> :
        /*A*/ 0x686b81b27164da8bf8eb0f4b70569d35<127:0> :
        /*9*/ 0x1e240e5e6358d1a225227c3b01217887<127:0> :
        /*8*/ 0xd40046579fd327524c3602e7a0c4c89e<127:0> :
        /*7*/ 0xeabf8ad240c738b5a3f7f2cef96115a1<127:0> :
        /*6*/ 0xe0ae5da49b341a55ad933230f58cb1e3<127:0> :
        /*5*/ 0x1df6e22e8266ca60c02923ab0d534e6f<127:0> :
        /*4*/ 0xd5db3745defd8e2f03ff6a726d6c5b51<127:0> :
        /*3*/ 0x8d1baf92bbddbc7f11d95c411f105ad8<127:0> :
        /*2*/ 0x0ac13188a5cd7bbd2d74d012b8e5b4b0<127:0> :
        /*1*/ 0x8969974a0c96777e65b9f109c56ec684<127:0> :
        /*0*/ 0x18f07dec3adc4d2079ee5f3ed7cb3948<127:0>
    );

    sboxout = sboxstring<(255-UInt(sboxin))*8+7:(255-UInt(sboxin))*8>;
    return sboxout;
```

## Library pseudocode for shared/functions/exclusive/ClearExclusiveByAddress

```
// ClearExclusiveByAddress()
// =====
// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.

ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);
```

## Library pseudocode for shared/functions/exclusive/ClearExclusiveLocal

```
// ClearExclusiveLocal()
// =====
// Clear the local Exclusives monitor for the specified processorid.

ClearExclusiveLocal(integer processorid);
```

## Library pseudocode for shared/functions/exclusive/ExclusiveMonitorsStatus

```
// ExclusiveMonitorsStatus()
// =====
// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.

bit ExclusiveMonitorsStatus();
```

### Library pseudocode for shared/functions/exclusive/IsExclusiveGlobal

```
// IsExclusiveGlobal()
// =====
// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.

boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

### Library pseudocode for shared/functions/exclusive/IsExclusiveLocal

```
// IsExclusiveLocal()
// =====
// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.

boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

### Library pseudocode for shared/functions/exclusive/MarkExclusiveGlobal

```
// MarkExclusiveGlobal()
// =====
// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.

MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

### Library pseudocode for shared/functions/exclusive/MarkExclusiveLocal

```
// MarkExclusiveLocal()
// =====
// Record the physical address region of size bytes starting at paddress in
// the local Exclusives monitor for processorid.

MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

### Library pseudocode for shared/functions/exclusive/ProcessorID

```
// ProcessorID()
// =====
// Return the ID of the currently executing PE.

integer ProcessorID();
```

### Library pseudocode for shared/functions/extension/AArch32.HaveHPDExt

```
// AArch32.HaveHPDExt()
// =====

boolean AArch32.HaveHPDExt()
    return IsFeatureImplemented(FEAT_AA32HPD);
```

### Library pseudocode for shared/functions/extension/AArch64.HaveHPDExt

```
// AArch64.HaveHPDExt()
// =====

boolean AArch64.HaveHPDExt()
    return IsFeatureImplemented(FEAT_HPDS);
```

### Library pseudocode for shared/functions/extension/Have128BitDescriptorExt

```
// Have128BitDescriptorExt()
// =====
// Returns TRUE if 128-bit Descriptor extension
// support is implemented and FALSE otherwise.

boolean Have128BitDescriptorExt()
    return IsFeatureImplemented(FEAT_D128);
```

### Library pseudocode for shared/functions/extension/Have16bitVMID

```
// Have16bitVMID()
// =====
// Returns TRUE if EL2 and support for a 16-bit VMID are implemented.

boolean Have16bitVMID()
    return IsFeatureImplemented(FEAT_VMID16);
```

### Library pseudocode for shared/functions/extension/Have52BitIPAAAndPASpaceExt

```
// Have52BitIPAAAndPASpaceExt()
// =====
// Returns TRUE if 52-bit IPA and PA extension support
// is implemented, and FALSE otherwise.

boolean Have52BitIPAAAndPASpaceExt()
    return IsFeatureImplemented(FEAT_LPA2);
```

### Library pseudocode for shared/functions/extension/Have52BitPAExt

```
// Have52BitPAExt()
// =====
// Returns TRUE if Large Physical Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitPAExt()
    return IsFeatureImplemented(FEAT_LPA);
```

### Library pseudocode for shared/functions/extension/Have52BitVAExt

```
// Have52BitVAExt()
// =====
// Returns TRUE if Large Virtual Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitVAExt()
    return IsFeatureImplemented(FEAT_LVA);
```

### Library pseudocode for shared/functions/extension/Have56BitPAExt

```
// Have56BitPAExt()
// =====
// Returns TRUE if 56-bit Physical Address extension
// support is implemented and FALSE otherwise.

boolean Have56BitPAExt()
    return IsFeatureImplemented(FEAT_D128);
```

### Library pseudocode for shared/functions/extension/Have56BitVAExt

```
// Have56BitVAExt()
// =====
// Returns TRUE if 56-bit Virtual Address extension
// support is implemented and FALSE otherwise.

boolean Have56BitVAExt()
    return IsFeatureImplemented(FEAT_LVA3);
```

### Library pseudocode for shared/functions/extension/HaveAArch32BF16Ext

```
// HaveAArch32BF16Ext()
// =====
// Returns TRUE if AArch32 BFloat16 instruction support is implemented, and FALSE otherwise.

boolean HaveAArch32BF16Ext()
    return IsFeatureImplemented(FEAT_AA32BF16);
```

### Library pseudocode for shared/functions/extension/HaveAArch32Int8MatMulExt

```
// HaveAArch32Int8MatMulExt()
// =====
// Returns TRUE if AArch32 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.

boolean HaveAArch32Int8MatMulExt()
    return IsFeatureImplemented(FEAT_AA32I8MM);
```

### Library pseudocode for shared/functions/extension/HaveAIEExt

```
// HaveAIEExt()
// =====
// Returns TRUE if AIE extension
// support is implemented and FALSE otherwise.

boolean HaveAIEExt()
    return IsFeatureImplemented(FEAT_AIE);
```

### Library pseudocode for shared/functions/extension/HaveAccessFlagUpdateExt

```
// HaveAccessFlagUpdateExt()
// =====

boolean HaveAccessFlagUpdateExt()
    return IsFeatureImplemented(FEAT_HAFDBS);
```

### Library pseudocode for shared/functions/extension/HaveAccessFlagUpdateForTableExt

```
// HaveAccessFlagUpdateForTableExt()
// =====
// Returns TRUE if support for Access Flag Update for Table Descriptors
// is implemented, and FALSE otherwise.

boolean HaveAccessFlagUpdateForTableExt()
    return IsFeatureImplemented(FEAT_HAFT);
```

### Library pseudocode for shared/functions/extension/HaveAltFP

```
// HaveAltFP()
// =====
// Returns TRUE if alternative Floating-point extension support
// is implemented, and FALSE otherwise.

boolean HaveAltFP()
    return IsFeatureImplemented(FEAT_AFP);
```

### Library pseudocode for shared/functions/extension/HaveAtomicExt

```
// HaveAtomicExt()
// =====

boolean HaveAtomicExt()
    return IsFeatureImplemented(FEAT_LSE);
```

### Library pseudocode for shared/functions/extension/HaveBF16Ext

```
// HaveBF16Ext()
// =====
// Returns TRUE if AArch64 BFloat16 instruction support is implemented, and FALSE otherwise.

boolean HaveBF16Ext()
    return IsFeatureImplemented(FEAT_BF16);
```

### Library pseudocode for shared/functions/extension/HaveBRBEv1p1

```
// HaveBRBEv1p1()
// =====
// Returns TRUE if BRBEv1p1 extension is implemented, and FALSE otherwise.

boolean HaveBRBEv1p1()
    return IsFeatureImplemented(FEAT_BRBEv1p1);
```

### Library pseudocode for shared/functions/extension/HaveBRBExt

```
// HaveBRBExt()
// =====
// Returns TRUE if Branch Record Buffer Extension is implemented, and FALSE otherwise.

boolean HaveBRBExt()
    return IsFeatureImplemented(FEAT_BRBE);
```

### Library pseudocode for shared/functions/extension/HaveBTIExt

```
// HaveBTIExt()
// =====
// Returns TRUE if support for Branch Target Identification is implemented.

boolean HaveBTIExt()
    return IsFeatureImplemented(FEAT_BTI);
```

### Library pseudocode for shared/functions/extension/HaveBlockBBM

```
// HaveBlockBBM()
// =====
// Returns TRUE if support for changing block size without requiring
// break-before-make is implemented.

boolean HaveBlockBBM()
    return IsFeatureImplemented(FEAT_BBM);
```



### Library pseudocode for shared/functions/extension/HaveCNTSCExt

```
// HaveCNTSCExt()
// =====
// Returns TRUE if the Generic Counter Scaling is implemented, and FALSE
// otherwise.

boolean HaveCNTSCExt()
    return IsFeatureImplemented(FEAT_CNTSC);
```

### Library pseudocode for shared/functions/extension/HaveCSSC

```
// HaveCSSC()
// =====
// Returns TRUE if the Common Short Sequence Compression instructions extension is implemented,
// and FALSE otherwise.

boolean HaveCSSC()
    return IsFeatureImplemented(FEAT_CSSC);
```

### Library pseudocode for shared/functions/extension/HaveCommonNotPrivateTransExt

```
// HaveCommonNotPrivateTransExt()
// =====

boolean HaveCommonNotPrivateTransExt()
    return IsFeatureImplemented(FEAT_TTCNP);
```

### Library pseudocode for shared/functions/extension/HaveDGHExt

```
// HaveDGHExt()
// =====
// Returns TRUE if Data Gathering Hint instruction support is implemented, and
// FALSE otherwise.

boolean HaveDGHExt()
    return IsFeatureImplemented(FEAT_DGH);
```

### Library pseudocode for shared/functions/extension/HaveDITExt

```
// HaveDITExt()
// =====

boolean HaveDITExt()
    return IsFeatureImplemented(FEAT_DIT);
```

### Library pseudocode for shared/functions/extension/HaveDOTPExt

```
// HaveDOTPExt()
// =====
// Returns TRUE if Dot Product feature support is implemented, and FALSE otherwise.

boolean HaveDOTPExt()
    return IsFeatureImplemented(FEAT_DotProd);
```

### Library pseudocode for shared/functions/extension/HaveDirtyBitModifierExt

```
// HaveDirtyBitModifierExt()
// =====

boolean HaveDirtyBitModifierExt()
    return IsFeatureImplemented(FEAT_HAFDBS);
```

### Library pseudocode for shared/functions/extension/HaveDoPD

```
// HaveDoPD()
// =====
// Returns TRUE if Debug Over Power Down extension
// support is implemented and FALSE otherwise.

boolean HaveDoPD()
    return IsFeatureImplemented(FEAT_DoPD);
```

### Library pseudocode for shared/functions/extension/HaveDoubleFault2Ext

```
// HaveDoubleFault2Ext()
// =====
// Returns TRUE if support for the DoubleFault2 feature is implemented, and FALSE otherwise.

boolean HaveDoubleFault2Ext()
    return IsFeatureImplemented(FEAT_DoubleFault2);
```

### Library pseudocode for shared/functions/extension/HaveDoubleFaultExt

```
// HaveDoubleFaultExt()
// =====

boolean HaveDoubleFaultExt()
    return IsFeatureImplemented(FEAT_DoubleFault);
```

### Library pseudocode for shared/functions/extension/HaveDoubleLock

```
// HaveDoubleLock()
// =====
// Returns TRUE if support for the OS Double Lock is implemented.

boolean HaveDoubleLock()
    return IsFeatureImplemented(FEAT_DoubleLock);
```

### Library pseudocode for shared/functions/extension/HaveE0PDExt

```
// HaveE0PDExt()
// =====
// Returns TRUE if support for constant fault times for unprivileged accesses
// to the memory map is implemented.

boolean HaveE0PDExt()
    return IsFeatureImplemented(FEAT_E0PD);
```

### Library pseudocode for shared/functions/extension/HaveEBF16

```
// HaveEBF16()
// =====
// Returns TRUE if the EBF16 extension is implemented, FALSE otherwise.

boolean HaveEBF16()
    return IsFeatureImplemented(FEAT_EBF16);
```

### Library pseudocode for shared/functions/extension/HaveECVExt

```
// HaveECVExt()
// =====
// Returns TRUE if Enhanced Counter Virtualization extension
// support is implemented, and FALSE otherwise.

boolean HaveECVExt()
    return IsFeatureImplemented(FEAT_ECV);
```

### Library pseudocode for shared/functions/extension/HaveETExt

```
// HaveETExt()
// =====
// Returns TRUE if Embedded Trace Extension is implemented, and FALSE otherwise.

boolean HaveETExt()
    return IsFeatureImplemented(FEAT_ETE);
```

### Library pseudocode for shared/functions/extension/HaveExtendedCacheSets

```
// HaveExtendedCacheSets()
// =====

boolean HaveExtendedCacheSets()
    return IsFeatureImplemented(FEAT_CCIDX);
```

### Library pseudocode for shared/functions/extension/HaveExtendedECDebugEvents

```
// HaveExtendedECDebugEvents()
// =====

boolean HaveExtendedECDebugEvents()
    return IsFeatureImplemented(FEAT_Debugv8p2);
```

### Library pseudocode for shared/functions/extension/HaveExtendedExecuteNeverExt

```
// HaveExtendedExecuteNeverExt()
// =====

boolean HaveExtendedExecuteNeverExt()
    return IsFeatureImplemented(FEAT_XNX);
```

### Library pseudocode for shared/functions/extension/HaveFCADDExt

```
// HaveFCADDExt()
// =====

boolean HaveFCADDExt()
    return IsFeatureImplemented(FEAT_FCMA);
```

### Library pseudocode for shared/functions/extension/HaveFGTExt

```
// HaveFGTExt()
// =====
// Returns TRUE if Fine-Grained Traps is implemented, and FALSE otherwise.

boolean HaveFGTExt()
    return IsFeatureImplemented(FEAT_FGT);
```

### Library pseudocode for shared/functions/extension/HaveJCVTZSExt

```
// HaveJCVTZSExt()
// =====

boolean HaveJCVTZSExt()
    return IsFeatureImplemented(FEAT_JSCVT);
```

### Library pseudocode for shared/functions/extension/HaveFP16MulNoRoundingToFP32Ext

```
// HaveFP16MulNoRoundingToFP32Ext()
// =====
// Returns TRUE if has FP16 multiply with no intermediate rounding accumulate
// to FP32 instructions, and FALSE otherwise

boolean HaveFP16MulNoRoundingToFP32Ext()
    return IsFeatureImplemented(FEAT_FHM);
```

### Library pseudocode for shared/functions/extension/HaveFeatABLE

```
// HaveFeatABLE()
// =====
// Returns TRUE if support for linking watchpoints to address matching
// breakpoints is implemented, and FALSE otherwise.

boolean HaveFeatABLE()
    return IsFeatureImplemented(FEAT_ABLE);
```

### Library pseudocode for shared/functions/extension/HaveFeatCLRBHB

```
// HaveFeatCLRBHB()
// =====
// Returns TRUE if the CLRBHB instruction is implemented, and FALSE otherwise.

boolean HaveFeatCLRBHB()
    return IsFeatureImplemented(FEAT_CLRBHB);
```

### Library pseudocode for shared/functions/extension/HaveFeatCMOW

```
// HaveFeatCMOW()
// =====
// Returns TRUE if the SCTL_R_EL1.CMOW bit is implemented and the SCTL_R_EL2.CMOW and
// HCRX_EL2.CMOW bits are implemented if EL2 is implemented.

boolean HaveFeatCMOW()
    return IsFeatureImplemented(FEAT_CMOW);
```

### Library pseudocode for shared/functions/extension/HaveFeatHBC

```
// HaveFeatHBC()
// =====
// Returns TRUE if the BC instruction is implemented, and FALSE otherwise.

boolean HaveFeatHBC()
    return IsFeatureImplemented(FEAT_HBC);
```

### Library pseudocode for shared/functions/extension/HaveFeatHCX

```
// HaveFeatHCX()
// =====
// Returns TRUE if HCRX_EL2 Trap Control register is implemented,
// and FALSE otherwise.

boolean HaveFeatHCX()
    return IsFeatureImplemented(FEAT_HCX);
```

### Library pseudocode for shared/functions/extension/HaveFeatHPMN0

```
// HaveFeatHPMN0()
// =====
// Returns TRUE if HDCR.HPMN or MDCR_EL2.HPMN is permitted to be 0 without
// generating UNPREDICTABLE behavior, and FALSE otherwise.

boolean HaveFeatHPMN0()
    return IsFeatureImplemented(FEAT_HPMN0);
```

### Library pseudocode for shared/functions/extension/HaveFeatLS64

```
// HaveFeatLS64()
// =====
// Returns TRUE if the LD64B, ST64B instructions are
// supported, and FALSE otherwise.

boolean HaveFeatLS64()
    return IsFeatureImplemented(FEAT_LS64);
```

### Library pseudocode for shared/functions/extension/HaveFeatLS64\_ACCDATA

```
// HaveFeatLS64_ACCDATA()
// =====
// Returns TRUE if the ST64BV0 instruction is
// supported, and FALSE otherwise.

boolean HaveFeatLS64_ACCDATA()
    return IsFeatureImplemented(FEAT_LS64_ACCDATA);
```

### Library pseudocode for shared/functions/extension/HaveFeatLS64\_V

```
// HaveFeatLS64_V()
// =====
// Returns TRUE if the ST64BV instruction is
// supported, and FALSE otherwise.

boolean HaveFeatLS64_V()
    return IsFeatureImplemented(FEAT_LS64_V);
```

### Library pseudocode for shared/functions/extension/HaveFeatMEC

```
// HaveFeatMEC()
// =====
// Returns TRUE if Memory Encryption Contexts are implemented, and FALSE otherwise.

boolean HaveFeatMEC()
    return IsFeatureImplemented(FEAT_MEC);
```

### Library pseudocode for shared/functions/extension/HaveFeatMOPS

```
// HaveFeatMOPS()
// =====
// Returns TRUE if the CPY* and SET* instructions are supported, and FALSE otherwise.

boolean HaveFeatMOPS()
    return IsFeatureImplemented(FEAT_MOPS);
```

### Library pseudocode for shared/functions/extension/HaveFeatNMI

```
// HaveFeatNMI()
// =====
// Returns TRUE if the Non-Maskable Interrupt extension is
// implemented, and FALSE otherwise.

boolean HaveFeatNMI()
    return IsFeatureImplemented(FEAT_NMI);
```

### Library pseudocode for shared/functions/extension/HaveFeatRPRES

```
// HaveFeatRPRES()
// =====
// Returns TRUE if reciprocal estimate implements 12-bit precision
// when FPCR.AH=1, and FALSE otherwise.

boolean HaveFeatRPRES()
    return IsFeatureImplemented(FEAT_RPRES);
```

### Library pseudocode for shared/functions/extension/HaveFeatSCTLR2

```
// HaveFeatSCTLR2()
// =====
// Returns TRUE if SCTLR2 extension
// support is implemented and FALSE otherwise.

boolean HaveFeatSCTLR2()
    return IsFeatureImplemented(FEAT_SCTLR2);
```

### Library pseudocode for shared/functions/extension/HaveFeatTCR2

```
// HaveFeatTCR2()
// =====
// Returns TRUE if TCR2 extension
// support is implemented and FALSE otherwise.

boolean HaveFeatTCR2()
    return IsFeatureImplemented(FEAT_TCR2);
```

### Library pseudocode for shared/functions/extension/HaveFeatTIDCP1

```
// HaveFeatTIDCP1()
// =====
// Returns TRUE if the SCTLR_EL1.TIDCP bit is implemented and the SCTLR_EL2.TIDCP bit
// is implemented if EL2 is implemented.

boolean HaveFeatTIDCP1()
    return IsFeatureImplemented(FEAT_TIDCP1);
```

### Library pseudocode for shared/functions/extension/HaveFeatTRBEEExt

```
// HaveFeatTRBEEExt()
// =====
// Returns TRUE if the Trace Buffer Extension external mode is implemented, and FALSE otherwise.

boolean HaveFeatTRBEEExt()
    return IsFeatureImplemented(FEAT_TRBE_EXT);
```

### Library pseudocode for shared/functions/extension/HaveFeatWFxT

```
// HaveFeatWFxT()
// =====
// Returns TRUE if WFET and WFIT instruction support is implemented,
// and FALSE otherwise.

boolean HaveFeatWFxT()
    return IsFeatureImplemented(FEAT_WFxT);
```

### Library pseudocode for shared/functions/extension/HaveFeatXS

```
// HaveFeatXS()
// =====
// Returns TRUE if XS attribute and the TLBI and DSB instructions with nXS qualifier
// are supported, and FALSE otherwise.

boolean HaveFeatXS()
    return IsFeatureImplemented(FEAT_XS);
```

### Library pseudocode for shared/functions/extension/HaveFlagFormatExt

```
// HaveFlagFormatExt()
// =====
// Returns TRUE if flag format conversion instructions implemented.

boolean HaveFlagFormatExt()
    return IsFeatureImplemented(FEAT_FlagM2);
```

### Library pseudocode for shared/functions/extension/HaveFlagManipulateExt

```
// HaveFlagManipulateExt()
// =====
// Returns TRUE if flag manipulate instructions are implemented.

boolean HaveFlagManipulateExt()
    return IsFeatureImplemented(FEAT_FlagM);
```

### Library pseudocode for shared/functions/extension/HaveFrintExt

```
// HaveFrintExt()
// =====
// Returns TRUE if FRINT instructions are implemented.

boolean HaveFrintExt()
    return IsFeatureImplemented(FEAT_FRINTTS);
```

### Library pseudocode for shared/functions/extension/HaveGCS

```
// HaveGCS()
// =====
// Returns TRUE if support for Guarded Control Stack is
// implemented, and FALSE otherwise.

boolean HaveGCS()
    return IsFeatureImplemented(FEAT_GCS);
```

### Library pseudocode for shared/functions/extension/HaveGTGExt

```
// HaveGTGExt()
// =====
// Returns TRUE if support for guest translation granule size is implemented.

boolean HaveGTGExt()
    return IsFeatureImplemented(FEAT_GTG);
```

### Library pseudocode for shared/functions/extension/HaveHPMDExt

```
// HaveHPMDExt()
// =====

boolean HaveHPMDExt()
    return IsFeatureImplemented(FEAT_PMUv3p1);
```

### Library pseudocode for shared/functions/extension/HaveIDSExt

```
// HaveIDSExt()
// =====
// Returns TRUE if ID register handling feature is implemented.

boolean HaveIDSExt()
    return IsFeatureImplemented(FEAT_IDST);
```

### Library pseudocode for shared/functions/extension/HaveIESB

```
// HaveIESB()
// =====

boolean HaveIESB()
    return IsFeatureImplemented(FEAT_IESB);
```

### Library pseudocode for shared/functions/extension/HaveInt8MatMulExt

```
// HaveInt8MatMulExt()
// =====
// Returns TRUE if AArch64 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.

boolean HaveInt8MatMulExt()
    return IsFeatureImplemented(FEAT_I8MM);
```

### Library pseudocode for shared/functions/extension/HaveLRCPC3Ext

```
// HaveLRCPC3Ext()
// =====
// Returns TRUE if FEAT_LRCPC3 instructions are supported, and FALSE otherwise.

boolean HaveLRCPC3Ext()
    return IsFeatureImplemented(FEAT_LRCPC3);
```

### Library pseudocode for shared/functions/extension/HaveLSE128

```
// HaveLSE128()
// =====
// Returns TRUE if LSE128 is implemented, and FALSE otherwise.

boolean HaveLSE128()
    return IsFeatureImplemented(FEAT_LSE128);
```

### Library pseudocode for shared/functions/extension/HaveLSE2Ext

```
// HaveLSE2Ext()
// =====
// Returns TRUE if LSE2 is implemented, and FALSE otherwise.

boolean HaveLSE2Ext()
    return IsFeatureImplemented(FEAT_LSE2);
```



### Library pseudocode for shared/functions/extension/HaveMPAMExt

```
// HaveMPAMExt()
// =====
// Returns TRUE if MPAM is implemented, and FALSE otherwise.

boolean HaveMPAMExt()
    return IsFeatureImplemented(FEAT_MPAM);
```

### Library pseudocode for shared/functions/extension/HaveMPAMv0p1Ext

```
// HaveMPAMv0p1Ext()
// =====
// Returns TRUE if MPAMv0p1 is implemented, and FALSE otherwise.

boolean HaveMPAMv0p1Ext()
    return IsFeatureImplemented(FEAT_MPAMv0p1);
```

### Library pseudocode for shared/functions/extension/HaveMPAMv1p1Ext

```
// HaveMPAMv1p1Ext()
// =====
// Returns TRUE if MPAMv1p1 is implemented, and FALSE otherwise.

boolean HaveMPAMv1p1Ext()
    return IsFeatureImplemented(FEAT_MPAMv1p1);
```

### Library pseudocode for shared/functions/extension/HaveMTE2Ext

```
// HaveMTE2Ext()
// =====
// Returns TRUE if MTE support is beyond EL0, and FALSE otherwise.

boolean HaveMTE2Ext()
    return IsFeatureImplemented(FEAT_MTE2);
```

### Library pseudocode for shared/functions/extension/HaveMTE4Ext

```
// HaveMTE4Ext()
// =====
// Returns TRUE if functionality in FEAT_MTE4 is implemented, and FALSE otherwise.

boolean HaveMTE4Ext()
    return IsFeatureImplemented(FEAT_MTE4);
```

### Library pseudocode for shared/functions/extension/HaveMTEAsymFaultExt

```
// HaveMTEAsymFaultExt()
// =====
// Returns TRUE if MTE Asymmetric Fault Handling support is
// implemented, and FALSE otherwise.

boolean HaveMTEAsymFaultExt()
    return IsFeatureImplemented(FEAT_MTE4);
```

### Library pseudocode for shared/functions/extension/HaveMTEAsyncExt

```
// HaveMTEAsyncExt()
// =====
// Returns TRUE if MTE supports Asynchronous faulting, and FALSE otherwise.

boolean HaveMTEAsyncExt()
    return IsFeatureImplemented(FEAT_MTE4);
```

### Library pseudocode for shared/functions/extension/HaveMTECanonicalTagCheckingExt

```
// HaveMTECanonicalTagCheckingExt()
// =====
// Returns TRUE if MTE Canonical Tag Checking functionality is
// implemented, and FALSE otherwise.

boolean HaveMTECanonicalTagCheckingExt()
    return IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS);
```

### Library pseudocode for shared/functions/extension/HaveMTEExt

```
// HaveMTEExt()
// =====
// Returns TRUE if instruction-only MTE implemented, and FALSE otherwise.

boolean HaveMTEExt()
    return IsFeatureImplemented(FEAT_MTE);
```

### Library pseudocode for shared/functions/extension/HaveMTEPermExt

```
// HaveMTEPermExt()
// =====
// Returns TRUE if MTE_PERM implemented, and FALSE otherwise.

boolean HaveMTEPermExt()
    return IsFeatureImplemented(FEAT_MTE_PERM);
```

### Library pseudocode for shared/functions/extension/HaveMTEStoreOnlyExt

```
// HaveMTEStoreOnlyExt()
// =====
// Returns TRUE if MTE Store-only Tag Checking functionality is
// implemented, and FALSE otherwise.

boolean HaveMTEStoreOnlyExt()
    return IsFeatureImplemented(FEAT_MTE_STORE_ONLY);
```

### Library pseudocode for shared/functions/extension/HaveNV2Ext

```
// HaveNV2Ext()
// =====
// Returns TRUE if Enhanced Nested Virtualization is implemented.

boolean HaveNV2Ext()
    return IsFeatureImplemented(FEAT_NV2);
```

### Library pseudocode for shared/functions/extension/HaveNVExt

```
// HaveNVExt()
// =====
// Returns TRUE if Nested Virtualization is implemented.

boolean HaveNVExt()
    return IsFeatureImplemented(FEAT_NV);
```

### Library pseudocode for shared/functions/extension/HaveNoSecurePMUDisableOverride

```
// HaveNoSecurePMUDisableOverride()
// =====

boolean HaveNoSecurePMUDisableOverride()
    return IsFeatureImplemented(FEAT_Debugv8p2);
```

### Library pseudocode for shared/functions/extension/HaveNoninvasiveDebugAuth

```
// HaveNoninvasiveDebugAuth()
// =====
// Returns TRUE if the Non-invasive debug controls are implemented.

boolean HaveNoninvasiveDebugAuth()
    return !IsFeatureImplemented(FEAT_Debugv8p4);
```

### Library pseudocode for shared/functions/extension/HavePAN3Ext

```
// HavePAN3Ext()
// =====
// Returns TRUE if SCTL_EL1.EPAN and SCTL_EL2.EPAN support is implemented,
// and FALSE otherwise.

boolean HavePAN3Ext()
    return IsFeatureImplemented(FEAT_PAN3);
```

### Library pseudocode for shared/functions/extension/HavePANExt

```
// HavePANExt()
// =====

boolean HavePANExt()
    return IsFeatureImplemented(FEAT_PAN);
```

### Library pseudocode for shared/functions/extension/HavePFAR

```
// HavePFAR()
// =====
// Returns TRUE if the Physical Fault Address Extension is implemented, and FALSE
// otherwise.

boolean HavePFAR()
    return IsFeatureImplemented(FEAT_PFAR);
```

### Library pseudocode for shared/functions/extension/HavePMUv3

```
// HavePMUv3()
// =====
// Returns TRUE if the Performance Monitors extension is implemented, and FALSE otherwise.

boolean HavePMUv3()
    return IsFeatureImplemented(FEAT_PMUv3);
```

### Library pseudocode for shared/functions/extension/HavePMUv3EDGE

```
// HavePMUv3EDGE()
// =====
// Returns TRUE if support for PMU event edge detection is implemented, and FALSE otherwise.

boolean HavePMUv3EDGE()
    return IsFeatureImplemented(FEAT_PMUv3_EDGE);
```

### Library pseudocode for shared/functions/extension/HavePMUv3ICNTR

```
// HavePMUv3ICNTR()
// =====
// Returns TRUE if support for the Fixed-function instruction counter is
// implemented, and FALSE otherwise.

boolean HavePMUv3ICNTR()
    return IsFeatureImplemented(FEAT_PMUv3_ICNTR);
```

### Library pseudocode for shared/functions/extension/HavePMUv3TH

```
// HavePMUv3TH()
// =====
// Returns TRUE if the PMUv3 threshold extension is implemented, and FALSE otherwise.

boolean HavePMUv3TH()
    return IsFeatureImplemented(FEAT_PMUv3_TH);
```

### Library pseudocode for shared/functions/extension/HavePMUv3p1

```
// HavePMUv3p1()
// =====
// Returns TRUE if the Performance Monitors extension is implemented, and FALSE otherwise.

boolean HavePMUv3p1()
    return IsFeatureImplemented(FEAT_PMUv3p1);
```

### Library pseudocode for shared/functions/extension/HavePMUv3p4

```
// HavePMUv3p4()
// =====
// Returns TRUE if the PMUv3.4 extension is implemented, and FALSE otherwise.

boolean HavePMUv3p4()
    return IsFeatureImplemented(FEAT_PMUv3p4);
```

### Library pseudocode for shared/functions/extension/HavePMUv3p5

```
// HavePMUv3p5()
// =====
// Returns TRUE if the PMUv3.5 extension is implemented, and FALSE otherwise.

boolean HavePMUv3p5()
    return IsFeatureImplemented(FEAT_PMUv3p5);
```

### Library pseudocode for shared/functions/extension/HavePMUv3p7

```
// HavePMUv3p7()
// =====
// Returns TRUE if the PMUv3.7 extension is implemented, and FALSE otherwise.

boolean HavePMUv3p7()
    return IsFeatureImplemented(FEAT_PMUv3p7);
```

### Library pseudocode for shared/functions/extension/HavePMUv3p9

```
// HavePMUv3p9()
// =====
// Returns TRUE if the PMUv3.9 extension is implemented, and FALSE otherwise.

boolean HavePMUv3p9()
    return IsFeatureImplemented(FEAT_PMUv3p9);
```

### Library pseudocode for shared/functions/extension/HavePageBasedHardwareAttributes

```
// HavePageBasedHardwareAttributes()
// =====

boolean HavePageBasedHardwareAttributes()
    return IsFeatureImplemented(FEAT_HPDS2);
```

### Library pseudocode for shared/functions/extension/HaveQRDMLAHExt

```
// HaveQRDMLAHExt ()
// =====

boolean HaveQRDMLAHExt ()
    return IsFeatureImplemented (FEAT_RDM);
```

### Library pseudocode for shared/functions/extension/HaveRASExt

```
// HaveRASExt ()
// =====

boolean HaveRASExt ()
    return IsFeatureImplemented (FEAT_RAS);
```

### Library pseudocode for shared/functions/extension/HaveRASv2Ext

```
// HaveRASv2Ext ()
// =====
// Returns TRUE if support for RASv2 is implemented, and FALSE otherwise.

boolean HaveRASv2Ext ()
    return IsFeatureImplemented (FEAT_RASv2);
```

### Library pseudocode for shared/functions/extension/HaveRME

```
// HaveRME ()
// =====
// Returns TRUE if the Realm Management Extension is implemented, and FALSE
// otherwise.

boolean HaveRME ()
    return IsFeatureImplemented (FEAT_RME);
```

### Library pseudocode for shared/functions/extension/HaveRNG

```
// HaveRNG ()
// =====
// Returns TRUE if Random Number Generator extension
// support is implemented and FALSE otherwise.

boolean HaveRNG ()
    return IsFeatureImplemented (FEAT_RNG);
```

### Library pseudocode for shared/functions/extension/HaveS1PIExt

```
// HaveS1PIExt ()
// =====
// Returns TRUE if the S1 Permission Indirection extension is
// implemented and FALSE otherwise.

boolean HaveS1PIExt ()
    return IsFeatureImplemented (FEAT_S1PIE);
```

### Library pseudocode for shared/functions/extension/HaveS1POExt

```
// HaveS1POExt ()
// =====
// Returns TRUE if the S1 Permission Overlay extension is
// implemented and FALSE otherwise.

boolean HaveS1POExt ()
    return IsFeatureImplemented (FEAT_S1POE);
```

### Library pseudocode for shared/functions/extension/HaveS2PIExt

```
// HaveS2PIExt()
// =====
// Returns TRUE if the S2 Permission Indirection extension is
// implemented and FALSE otherwise.

boolean HaveS2PIExt()
    return IsFeatureImplemented(FEAT_S2PIE);
```

### Library pseudocode for shared/functions/extension/HaveS2POExt

```
// HaveS2POExt()
// =====
// Returns TRUE if the S2 Permission Overlay extension is
// implemented and FALSE otherwise.

boolean HaveS2POExt()
    return IsFeatureImplemented(FEAT_S2POE);
```

### Library pseudocode for shared/functions/extension/HaveSBExt

```
// HaveSBExt()
// =====
// Returns TRUE if support for SB is implemented, and FALSE otherwise.

boolean HaveSBExt()
    return IsFeatureImplemented(FEAT_SB);
```

### Library pseudocode for shared/functions/extension/HaveSSBSExt

```
// HaveSSBSExt()
// =====
// Returns TRUE if support for SSBS is implemented, and FALSE otherwise.

boolean HaveSSBSExt()
    return IsFeatureImplemented(FEAT_SSBS);
```

### Library pseudocode for shared/functions/extension/HaveSecureEL2Ext

```
// HaveSecureEL2Ext()
// =====
// Returns TRUE if Secure EL2 is implemented.

boolean HaveSecureEL2Ext()
    return IsFeatureImplemented(FEAT_SEL2);
```

### Library pseudocode for shared/functions/extension/HaveSecureExtDebugView

```
// HaveSecureExtDebugView()
// =====
// Returns TRUE if support for Secure and Non-secure views of debug peripherals
// is implemented.

boolean HaveSecureExtDebugView()
    return IsFeatureImplemented(FEAT_Debugv8p4);
```

### Library pseudocode for shared/functions/extension/HaveSelfHostedTrace

```
// HaveSelfHostedTrace()
// =====

boolean HaveSelfHostedTrace()
    return IsFeatureImplemented(FEAT_TRF);
```

### Library pseudocode for shared/functions/extension/HaveSmallTranslationTblExt

```
// HaveSmallTranslationTblExt()
// =====
// Returns TRUE if Small Translation Table Support is implemented.

boolean HaveSmallTranslationTableExt()
    return IsFeatureImplemented(FEAT_TTST);
```

### Library pseudocode for shared/functions/extension/HaveSoftwareLock

```
// HaveSoftwareLock()
// =====
// Returns TRUE if Software Lock is implemented.

boolean HaveSoftwareLock(Component component)
    if Havev8p4Debug() then
        return FALSE;
    if HaveDoPD() && component != Component\_CTI then
        return FALSE;
    case component of
        when Component\_Debug
            return boolean IMPLEMENTATION_DEFINED "Debug has Software Lock";
        when Component\_PMU
            return boolean IMPLEMENTATION_DEFINED "PMU has Software Lock";
        when Component\_CTI
            return boolean IMPLEMENTATION_DEFINED "CTI has Software Lock";
        otherwise
            Unreachable();
```

### Library pseudocode for shared/functions/extension/HaveStage2MemAttrControl

```
// HaveStage2MemAttrControl()
// =====
// Returns TRUE if support for Stage2 control of memory types and cacheability
// attributes is implemented.

boolean HaveStage2MemAttrControl()
    return IsFeatureImplemented(FEAT_S2FWB);
```

### Library pseudocode for shared/functions/extension/HaveStatisticalProfiling

```
// HaveStatisticalProfiling()
// =====
// Returns TRUE if Statistical Profiling Extension is implemented,
// and FALSE otherwise.

boolean HaveStatisticalProfiling()
    return IsFeatureImplemented(FEAT_SPE);
```

### Library pseudocode for shared/functions/extension/HaveStatisticalProfilingFDS

```
// HaveStatisticalProfilingFDS()
// =====
// Returns TRUE if the SPE_FDS extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingFDS()
    return IsFeatureImplemented(FEAT_SPE_FDS);
```

### Library pseudocode for shared/functions/extension/HaveStatisticalProfilingv1p1

```
// HaveStatisticalProfilingv1p1()
// =====
// Returns TRUE if the SPEv1p1 extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingv1p1()
    return IsFeatureImplemented(FEAT_SPEv1p1);
```

### Library pseudocode for shared/functions/extension/HaveStatisticalProfilingv1p2

```
// HaveStatisticalProfilingv1p2()
// =====
// Returns TRUE if the SPEv1p2 extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingv1p2()
    return IsFeatureImplemented(FEAT_SPEv1p2);
```

### Library pseudocode for shared/functions/extension/HaveStatisticalProfilingv1p4

```
// HaveStatisticalProfilingv1p4()
// =====
// Returns TRUE if the SPEv1p4 extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingv1p4()
    return IsFeatureImplemented(FEAT_SPEv1p4);
```

### Library pseudocode for shared/functions/extension/HaveSysInstr128

```
// HaveSysInstr128()
// =====
// Returns TRUE if support for System Instructions that can
// take 128-bit inputs is implemented, and FALSE otherwise.

boolean HaveSysInstr128()
    return IsFeatureImplemented(FEAT_SYSINSTR128);
```

### Library pseudocode for shared/functions/extension/HaveSysReg128

```
// HaveSysReg128()
// =====
// Returns TRUE if support for 128-bit System Registers is implemented, and FALSE otherwise.

boolean HaveSysReg128()
    return IsFeatureImplemented(FEAT_SYSREG128);
```

### Library pseudocode for shared/functions/extension/HaveTHExt

```
// HaveTHExt()
// =====
// Returns TRUE if support for Translation Hardening Extension is implemented.

boolean HaveTHExt()
    return IsFeatureImplemented(FEAT_THE);
```

### Library pseudocode for shared/functions/extension/HaveTME

```
// HaveTME()
// =====

boolean HaveTME()
    return IsFeatureImplemented(FEAT_TME);
```



### Library pseudocode for shared/functions/extension/HaveTWEDExt

```
// HaveTWEDExt ()
// =====
// Returns TRUE if Delayed Trapping of WFE instruction support is implemented,
// and FALSE otherwise.

boolean HaveTWEDExt ()
    return IsFeatureImplemented (FEAT_TWED);
```

### Library pseudocode for shared/functions/extension/HaveTraceBufferExtension

```
// HaveTraceBufferExtension ()
// =====
// Returns TRUE if Trace Buffer Extension is implemented, and FALSE otherwise.

boolean HaveTraceBufferExtension ()
    return IsFeatureImplemented (FEAT_TRBE);
```

### Library pseudocode for shared/functions/extension/HaveTraceExt

```
// HaveTraceExt ()
// =====
// Returns TRUE if Trace functionality as described by the Trace Architecture
// is implemented.

boolean HaveTraceExt ()
    return boolean IMPLEMENTATION_DEFINED "Has Trace Architecture functionality";
```

### Library pseudocode for shared/functions/extension/HaveTrapLoadStoreMultipleDeviceExt

```
// HaveTrapLoadStoreMultipleDeviceExt ()
// =====

boolean HaveTrapLoadStoreMultipleDeviceExt ()
    return IsFeatureImplemented (FEAT_LSMAOC);
```

### Library pseudocode for shared/functions/extension/HaveUAOExt

```
// HaveUAOExt ()
// =====

boolean HaveUAOExt ()
    return IsFeatureImplemented (FEAT_UAO);
```

### Library pseudocode for shared/functions/extension/HaveV82Debug

```
// HaveV82Debug ()
// =====

boolean HaveV82Debug ()
    return IsFeatureImplemented (FEAT_Debugv8p2);
```

### Library pseudocode for shared/functions/extension/HaveVirtHostExt

```
// HaveVirtHostExt ()
// =====

boolean HaveVirtHostExt ()
    return IsFeatureImplemented (FEAT_VHE);
```

### Library pseudocode for shared/functions/extension/Havev8p4Debug

```
// Havev8p4Debug()
// =====
// Returns TRUE if support for the Debugv8p4 feature is implemented and FALSE otherwise.

boolean Havev8p4Debug()
    return IsFeatureImplemented(FEAT_Debugv8p4);
```

### Library pseudocode for shared/functions/extension/Havev8p8Debug

```
// Havev8p8Debug()
// =====
// Returns TRUE if support for the Debugv8p8 feature is implemented and FALSE otherwise.

boolean Havev8p8Debug()
    return IsFeatureImplemented(FEAT_Debugv8p8);
```

### Library pseudocode for shared/functions/extension/Havev8p9Debug

```
// Havev8p9Debug()
// =====
// Returns TRUE if support for the Debugv8p9 feature is implemented, and FALSE otherwise.

boolean Havev8p9Debug()
    return IsFeatureImplemented(FEAT_Debugv8p9);
```

### Library pseudocode for shared/functions/extension/InsertIESBBeforeException

```
// InsertIESBBeforeException()
// =====
// Returns an implementation defined choice whether to insert an implicit error synchronization
// barrier before exception.
// If SCTLR_ELx.IESB is 1 when an exception is generated to ELx, any pending Unrecoverable
// SError interrupt must be taken before executing any instructions in the exception handler.
// However, this can be before the branch to the exception handler is made.

boolean InsertIESBBeforeException(bits(2) el)
    return (HaveIESB\(\) && boolean IMPLEMENTATION_DEFINED
        "Has Implicit Error Synchronization Barrier before Exception");
```

### Library pseudocode for shared/functions/extension/IsG1ActivityMonitorImplemented

```
// IsG1ActivityMonitorImplemented()
// =====
// Returns TRUE if a G1 activity monitor is implemented for the counter
// and FALSE otherwise.

boolean IsG1ActivityMonitorImplemented(integer i);
```

### Library pseudocode for shared/functions/extension/IsG1ActivityMonitorOffsetImplemented

```
// IsG1ActivityMonitorOffsetImplemented()
// =====
// Returns TRUE if a G1 activity monitor offset is implemented for the counter,
// and FALSE otherwise.

boolean IsG1ActivityMonitorOffsetImplemented(integer i);
```

## Library pseudocode for shared/functions/externalaborts/AArch32.PEErrorState

```
// AArch32.PEErrorState()
// =====
// Returns the error state by PE on taking an SError Interrupt
// to AArch32 level.

ErrorState AArch32.PEErrorState(FaultRecord fault)
    if (!ErrorIsContained() ||
        (!ErrorIsSynchronized() && !StateIsRecoverable()) ||
        ReportErrorAsUC()) then
        return ErrorState_UC;

    if !StateIsRecoverable() || ReportErrorAsUEU() then
        return ErrorState_UEU;

    if ActionRequired() || ReportErrorAsUER() then
        return ErrorState_UER;

    return ErrorState_UEO;
```

## Library pseudocode for shared/functions/externalaborts/AArch64.PEErrorState

```
// AArch64.PEErrorState()
// =====
// Returns the error state by PE on taking a Synchronous
// or Asynchronous exception.

ErrorState AArch64.PEErrorState(FaultRecord fault)
    if !IsExternalSyncAbort(fault) && ExtAbortToA64(fault) then
        if ReportErrorAsUncategorized() then
            return ErrorState_Uncategorized;
        if ReportErrorAsIMPDEF() then
            return ErrorState_IMPDEF;

    assert !FaultIsCorrected();
    if (!ErrorIsContained() ||
        (!ErrorIsSynchronized() && !StateIsRecoverable()) ||
        ReportErrorAsUC()) then
        return ErrorState_UC;

    if !StateIsRecoverable() || ReportErrorAsUEU() then
        if IsExternalSyncAbort(fault) then // Implies taken to AArch64
            return ErrorState_UC;
        else
            return ErrorState_UEU;

    if (ActionRequired() || ReportErrorAsUER()) then
        return ErrorState_UER;

    return ErrorState_UEO;
```

## Library pseudocode for shared/functions/externalaborts/ActionRequired

```
// ActionRequired()
// =====
// Return an implementation specific value:
// returns TRUE if action is required, FALSE otherwise.

boolean ActionRequired();
```

## Library pseudocode for shared/functions/externalaborts/ClearPendingPhysicalSError

```
// ClearPendingPhysicalSError()
// =====
// Clear a pending physical SError interrupt.

ClearPendingPhysicalSError();
```

## Library pseudocode for shared/functions/externalaborts/ClearPendingVirtualSError

```
// ClearPendingVirtualSError()
// =====
// Clear a pending virtual SError interrupt.

ClearPendingVirtualSError()
    if ELUsingAArch32(EL2) then
        HCR.VA = '0';
    else
        HCR_EL2.VSE = '0';
```

## Library pseudocode for shared/functions/externalaborts/ErrorIsContained

```
// ErrorIsContained()
// =====
// Return an implementation specific value:
// TRUE if Error is contained by the PE, FALSE otherwise.

boolean ErrorIsContained();
```

## Library pseudocode for shared/functions/externalaborts/ErrorIsSynchronized

```
// ErrorIsSynchronized()
// =====
// Return an implementation specific value:
// returns TRUE if Error is synchronized by any synchronization event
// FALSE otherwise.

boolean ErrorIsSynchronized();
```

## Library pseudocode for shared/functions/externalaborts/ExtAbortToA64

```
// ExtAbortToA64()
// =====
// Returns TRUE if synchronous exception is being taken to A64 exception
// level.

boolean ExtAbortToA64(FaultRecord fault)
    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
            (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
            (IsDebugException(fault) && MDCR_EL2.TDE == '1'));

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR\_curr[].EA == '1' && IsExternalAbort(fault);

    return route_to_aarch64 && IsExternalSyncAbort(fault.statuscode);
```

## Library pseudocode for shared/functions/externalaborts/FaultIsCorrected

```
// FaultIsCorrected()
// =====
// Return an implementation specific value:
// TRUE if fault is corrected by the PE, FALSE otherwise.

boolean FaultIsCorrected();
```

## Library pseudocode for shared/functions/externalaborts/GetPendingPhysicalSError

```
// GetPendingPhysicalSError()
// =====
// Returns the FaultRecord containing details of pending Physical SError
// interrupt.

FaultRecord GetPendingPhysicalSError();
```

## Library pseudocode for shared/functions/externalaborts/HandleExternalAbort

```
// HandleExternalAbort()
// =====
// Takes a Synchronous/Asynchronous abort based on fault.

HandleExternalAbort(PhysMemRetStatus memretstatus, boolean iswrite,
                   AddressDescriptor memaddrdesc, integer size,
                   AccessDescriptor accdesc)
    assert (memretstatus.statuscode IN {Fault\_SyncExternal, Fault\_AsyncExternal} ||
           (!HaveRASExt() && memretstatus.statuscode IN {Fault\_SyncParity,
                                                         Fault\_AsyncParity}));

    fault = NoFault(accdesc);
    fault.statuscode = memretstatus.statuscode;
    fault.write      = iswrite;
    fault.extflag    = memretstatus.extflag;
    // It is implementation specific whether External aborts signaled
    // in-band synchronously are taken synchronously or asynchronously
    if (IsExternalSyncAbort(fault) &&
        !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
                                           size, accdesc)) then
        if fault.statuscode == Fault\_SyncParity then
            fault.statuscode = Fault\_AsyncParity;
        else
            fault.statuscode = Fault\_AsyncExternal;

    if HaveRASExt() then
        fault.merrorstate = memretstatus.merrorstate;

    if IsExternalSyncAbort(fault) then
        if UsingAArch32() then
            AArch32.Abort(memaddrdesc.vaddress<31:0>, fault);
        else
            AArch64.Abort(memaddrdesc.vaddress, fault);

    else
        PendSErrorInterrupt(fault);
```

## Library pseudocode for shared/functions/externalaborts/HandleExternalReadAbort

```
// HandleExternalReadAbort()
// =====
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory read.

HandleExternalReadAbort(PhysMemRetStatus memstatus, AddressDescriptor memaddrdesc,
                       integer size, AccessDescriptor accdesc)
    iswrite = FALSE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
```

## Library pseudocode for shared/functions/externalaborts/HandleExternalTTWAbort

```
// HandleExternalTTWAbort()
// =====
// Take Asynchronous abort or update FaultRecord for Translation Table Walk
// based on PhysMemRetStatus.

FaultRecord HandleExternalTTWAbort(PhysMemRetStatus memretstatus, boolean iswrite,
                                   AddressDescriptor memaddrdesc,
                                   AccessDescriptor accdesc, integer size,
                                   FaultRecord input_fault)

    output_fault = input_fault;
    output_fault.extflag = memretstatus.extflag;
    output_fault.statuscode = memretstatus.statuscode;
    if (IsExternalSyncAbort(output_fault) &&
        !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
                                           size, accdesc)) then
        if output_fault.statuscode == Fault\_SyncParity then
            output_fault.statuscode = Fault\_AsyncParity;
        else
            output_fault.statuscode = Fault\_AsyncExternal;

// If a synchronous fault is on a translation table walk, then update
// the fault type
if IsExternalSyncAbort(output_fault) then
    if output_fault.statuscode == Fault\_SyncParity then
        output_fault.statuscode = Fault\_SyncParityOnWalk;
    else
        output_fault.statuscode = Fault\_SyncExternalOnWalk;
if HaveRASExt() then
    output_fault.merrorstate = memretstatus.merrorstate;
if !IsExternalSyncAbort(output_fault) then
    PendErrorInterrupt(output_fault);
    output_fault.statuscode = Fault\_None;
return output_fault;
```

## Library pseudocode for shared/functions/externalaborts/HandleExternalWriteAbort

```
// HandleExternalWriteAbort()
// =====
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory write.

HandleExternalWriteAbort(PhysMemRetStatus memstatus, AddressDescriptor memaddrdesc,
                        integer size, AccessDescriptor accdesc)
    iswrite = TRUE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
```

## Library pseudocode for shared/functions/externalaborts/IsExternalAbortTakenSynchronously

```
// IsExternalAbortTakenSynchronously()
// =====
// Return an implementation specific value:
// TRUE if the fault returned for the access can be taken synchronously,
// FALSE otherwise.
//
// This might vary between accesses, for example depending on the error type
// or memory type being accessed.
// External aborts on data accesses and translation table walks on data accesses
// can be either synchronous or asynchronous.
//
// When FEAT_DoubleFault is not implemented, External aborts on instruction
// fetches and translation table walks on instruction fetches can be either
// synchronous or asynchronous.
// When FEAT_DoubleFault is implemented, all External abort exceptions on
// instruction fetches and translation table walks on instruction fetches
// must be synchronous.

boolean IsExternalAbortTakenSynchronously(PhysMemRetStatus memstatus,
                                         boolean iswrite,
                                         AddressDescriptor desc,
                                         integer size,
                                         AccessDescriptor accdesc);
```

## Library pseudocode for shared/functions/externalaborts/IsPhysicalSErrorPending

```
// IsPhysicalSErrorPending()
// =====
// Returns TRUE if a physical SError interrupt is pending.

boolean IsPhysicalSErrorPending();
```

## Library pseudocode for shared/functions/externalaborts/IsSErrorEdgeTriggered

```
// IsSErrorEdgeTriggered()
// =====
// Returns TRUE if the physical SError interrupt is edge-triggered
// and FALSE otherwise.

boolean IsSErrorEdgeTriggered()
    if HaveDoubleFaultExt() then
        return TRUE;
    else
        return boolean IMPLEMENTATION_DEFINED "Edge-triggered SError";
```

## Library pseudocode for shared/functions/externalaborts/IsSynchronizablePhysicalSErrorPending

```
// IsSynchronizablePhysicalSErrorPending()
// =====
// Returns TRUE if a synchronizable physical SError interrupt is pending.

boolean IsSynchronizablePhysicalSErrorPending();
```

## Library pseudocode for shared/functions/externalaborts/IsVirtualSErrorPending

```
// IsVirtualSErrorPending()
// =====
// Return TRUE if a virtual SError interrupt is pending.

boolean IsVirtualSErrorPending()
    if ELUsingAArch32(EL2) then
        return HCR.VA == '1';
    else
        return HCR_EL2.VSE == '1';
```

### Library pseudocode for shared/functions/externalaborts/PendSErrorInterrupt

```
// PendSErrorInterrupt()
// =====
// Pend the SError Interrupt.

PendSErrorInterrupt(FaultRecord fault);
```

### Library pseudocode for shared/functions/externalaborts/ReportErrorAsIMPDEF

```
// ReportErrorAsIMPDEF()
// =====
// Return an implementation specific value:
// returns TRUE if Error is IMPDEF, FALSE otherwise.

boolean ReportErrorAsIMPDEF();
```

### Library pseudocode for shared/functions/externalaborts/ReportErrorAsUC

```
// ReportErrorAsUC()
// =====
// Return an implementation specific value:
// returns TRUE if Error is Uncontainable, FALSE otherwise.

boolean ReportErrorAsUC();
```

### Library pseudocode for shared/functions/externalaborts/ReportErrorAsUER

```
// ReportErrorAsUER()
// =====
// Return an implementation specific value:
// returns TRUE if Error is Recoverable, FALSE otherwise.

boolean ReportErrorAsUER();
```

### Library pseudocode for shared/functions/externalaborts/ReportErrorAsUEU

```
// ReportErrorAsUEU()
// =====
// Return an implementation specific value:
// returns TRUE if Error is Unrecoverable, FALSE otherwise.

boolean ReportErrorAsUEU();
```

### Library pseudocode for shared/functions/externalaborts/ReportErrorAsUncategorized

```
// ReportErrorAsUncategorized()
// =====
// Return an implementation specific value:
// returns TRUE if Error is uncategorized, FALSE otherwise.

boolean ReportErrorAsUncategorized();
```

### Library pseudocode for shared/functions/externalaborts/StateIsRecoverable

```
// StateIsRecoverable()
// =====
// Return an implementation specific value:
// returns TRUE is PE State is unrecoverable else FALSE.

boolean StateIsRecoverable();
```



## Library pseudocode for shared/functions/float/bfloat/BFAdd

```
// BFAdd()
// =====
// Non-widening BFloat16 addition used by SVE2 instructions.

bits(N) BFAdd(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean fpexc = TRUE;
    return BFAdd(op1, op2, fpcr, fpexc);

// BFAdd()
// =====
// Non-widening BFloat16 addition following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates op1 + op2.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFAdd(bits(N) op1, bits(N) op2, FPCRTType fpcr, boolean fpexc)

    assert N == 16;
    FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    bits(2*N) op1_s = op1 : Zeros(N);
    bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);

    if !done then
        inf1 = (type1 == FPTType Infinity);
        inf2 = (type2 == FPTType Infinity);
        zero1 = (type1 == FPTType Zero);
        zero2 = (type2 == FPTType Zero);

        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPExc InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0', 2*N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1', 2*N);
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, 2*N);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding NEGINF then '1' else '0';
                result = FPZero(result_sign, 2*N);
            else
                result = FPRoundBF(result_value, fpcr, rounding, fpexc, 2*N);

            if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```

## Library pseudocode for shared/functions/float/bfloat/BFAdd\_ZA

```
// BFAdd_ZA()
// =====
// Non-widening BFloat16 addition used by SME2 ZA-targeting instructions.

bits(N) BFAdd_ZA(bits(N) op1, bits(N) op2, FPCRTType fpcr_in)
    boolean fpexc = FALSE;
    FPCRTType fpcr = fpcr_in;
    fpcr.DN = '1'; // Generate default NaN values
    return BFAdd(op1, op2, fpcr, fpexc);
```

## Library pseudocode for shared/functions/float/bfloat/BFDotAdd

```
// BFDotAdd()
// =====
// BFloat16 2-way dot-product and add to single-precision
// result = addend + op1_a*op2_a + op1_b*op2_b

bits(N) BFDotAdd(bits(N) addend, bits(N DIV 2) op1_a, bits(N DIV 2) op1_b,
                 bits(N DIV 2) op2_a, bits(N DIV 2) op2_b, FPCRTType fpcr_in)
    assert N == 32;
    FPCRTType fpcr = fpcr_in;

    bits(N) prod;

    bits(N) result;
    if !HaveEBF16() || fpcr.EBF == '0' then // Standard BFloat16 behaviors
        prod = FPAdd\_BF16(BFMulH(op1_a, op2_a), BFMulH(op1_b, op2_b));
        result = FPAdd\_BF16(addend, prod);
    else // Extended BFloat16 behaviors
        boolean isbfloat16 = TRUE;
        boolean fpexc = FALSE; // Do not generate floating-point exceptions
        fpcr.DN = '1'; // Generate default NaN values
        prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc, N);
        result = FPAdd(addend, prod, fpcr, fpexc);

    return result;
```

## Library pseudocode for shared/functions/float/bfloat/BFInfinity

```
// BFInfinity()
// =====

bits(N) BFInfinity(bit sign, integer N)
    assert N == 16;
    constant integer E = 8;
    constant integer F = N - (E + 1);
    return sign : Ones(E) : Zeros(F);
```

## Library pseudocode for shared/functions/float/bfloat/BFMatMulAdd

```
// BFMatMulAdd()
// =====
// BFloat16 matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])

bits(N) BFMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2)

    assert N == 128;

    bits(N) result;
    bits(32) sum;

    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, 32];
            for k = 0 to 1
                bits(16) elt1_a = Elem[op1, 4*i + 2*k + 0, 16];
                bits(16) elt1_b = Elem[op1, 4*i + 2*k + 1, 16];
                bits(16) elt2_a = Elem[op2, 4*j + 2*k + 0, 16];
                bits(16) elt2_b = Elem[op2, 4*j + 2*k + 1, 16];
                sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR);
                Elem[result, 2*i + j, 32] = sum;

    return result;
```

## Library pseudocode for shared/functions/float/bfloat/BFMax

```
// BFMax()
// =====
// BFloat16 maximum.

bits(N) BFMax(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return BFMax(op1, op2, fpcr, altfp);

// BFMax()
// =====
// BFloat16 maximum following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the larger value after rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behavior.

bits(N) BFMax(bits(N) op1, bits(N) op2, FPCRTType fpcr_in, boolean altfp)

    assert N == 16;
    FPCRTType fpcr = fpcr_in;
    boolean fpexc = TRUE;
    FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    bits(2*N) op1_s = op1 : Zeros(N);
    bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    if altfp && type1 == FPTYPE\_Zero && type2 == FPTYPE\_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return BFZero(sign2, N);
    elsif altfp && (type1 IN {FPTYPE\_SNaN, FPTYPE\_QNaN} || type2 IN {FPTYPE\_SNaN, FPTYPE\_QNaN}) then
        // Alternate handling of NaN inputs
        FPProcessException(FPExc\_InvalidOp, fpcr);
        return (if type2 == FPTYPE\_Zero then BFZero(sign2, N) else op2);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr);
    if !done then
        FPTYPE fptype;
        bit sign;
        real value;
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPTYPE\_Infinity then
            result = FPInfinity(sign, 2*N);
        elsif fptype == FPTYPE\_Zero then
            sign = sign1 AND sign2; // Use most positive sign
            result = FPZero(sign, 2*N);
        else
            if altfp then // Denormal output is not flushed to zero
                fpcr.FZ = '0';
            result = FPRoundBF(value, fpcr, rounding, fpexc, 2*N);

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```

## Library pseudocode for shared/functions/float/bfloat/BFMaxNum

```
// BFMaxNum()
// =====
// BFloat16 maximum number following computational behaviors corresponding
// to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the smaller number operand after rounding.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFMaxNum(bits(N) op1_in, bits(N) op2_in, FPCRTYPE fpcr)

    assert N == 16;
    boolean fpexc = TRUE;
    boolean isbfloat16 = TRUE;
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    bits(N) result;

    (type1,-,-) = FPUnpackBase(op1, fpcr, fpexc, isbfloat16);
    (type2,-,-) = FPUnpackBase(op2, fpcr, fpexc, isbfloat16);

    boolean type1_nan = type1 IN {FPTYPE\_QNaN, FPTYPE\_SNaN};
    boolean type2_nan = type2 IN {FPTYPE\_QNaN, FPTYPE\_SNaN};

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as -Infinity.
        if type1 == FPTYPE\_QNaN && type2 != FPTYPE\_QNaN then
            op1 = BFInfinity('1', N);
        elsif type1 != FPTYPE\_QNaN && type2 == FPTYPE\_QNaN then
            op2 = BFInfinity('1', N);

    boolean altfmaxfmin = FALSE;    // Do not use alternate NaN handling
    result = BFMax(op1, op2, fpcr, altfmaxfmin);

    return result;
```

## Library pseudocode for shared/functions/float/bfloat/BFMin

```
// BFMin()
// =====
// BFloat16 minimum.

bits(N) BFMin(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return BFMin(op1, op2, fpcr, altfp);

// BFMin()
// =====
// BFloat16 minimum following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the smaller value after rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behavior.

bits(N) BFMin(bits(N) op1, bits(N) op2, FPCRTType fpcr_in, boolean altfp)

    assert N == 16;
    FPCRTType fpcr = fpcr_in;
    boolean fpexc = TRUE;
    FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    bits(2*N) op1_s = op1 : Zeros(N);
    bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    if altfp && type1 == FPTYPE\_Zero && type2 == FPTYPE\_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return BFZero(sign2, N);
    elsif altfp && (type1 IN {FPTYPE\_SNaN, FPTYPE\_QNaN} || type2 IN {FPTYPE\_SNaN, FPTYPE\_QNaN}) then
        // Alternate handling of NaN inputs
        FPProcessException(FPExc\_InvalidOp, fpcr);
        return (if type2 == FPTYPE\_Zero then BFZero(sign2, N) else op2);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr);
    if !done then
        FPTYPE fptype;
        bit sign;
        real value;
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPTYPE\_Infinity then
            result = FPInfinity(sign, 2*N);
        elsif fptype == FPTYPE\_Zero then
            sign = sign1 OR sign2; // Use most negative sign
            result = FPZero(sign, 2*N);
        else
            if altfp then // Denormal output is not flushed to zero
                fpcr.FZ = '0';
            result = FPRoundBF(value, fpcr, rounding, fpexc, 2*N);

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```

## Library pseudocode for shared/functions/float/bfloat/BFMinNum

```
// BFMinNum()
// =====
// BFloat16 minimum number following computational behaviors corresponding
// to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the smaller number operand after rounding.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFMinNum(bits(N) op1_in, bits(N) op2_in, FPCRTYPE fpcr)

    assert N == 16;
    boolean fpexc = TRUE;
    boolean isbfloat16 = TRUE;
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    bits(N) result;

    (type1,-,-) = FPUnpackBase(op1, fpcr, fpexc, isbfloat16);
    (type2,-,-) = FPUnpackBase(op2, fpcr, fpexc, isbfloat16);

    boolean type1_nan = type1 IN {FPTYPE\_QNaN, FPTYPE\_SNaN};
    boolean type2_nan = type2 IN {FPTYPE\_QNaN, FPTYPE\_SNaN};

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as +Infinity.
        if type1 == FPTYPE\_QNaN && type2 != FPTYPE\_QNaN then
            op1 = BFInfinity('0', N);
        elsif type1 != FPTYPE\_QNaN && type2 == FPTYPE\_QNaN then
            op2 = BFInfinity('0', N);

    boolean altfmaxfmin = FALSE;    // Do not use alternate NaN handling
    result = BFMin(op1, op2, fpcr, altfmaxfmin);

    return result;
```

## Library pseudocode for shared/functions/float/bfloat/BFMul

```
// BFMul()
// =====
// Non-widening BFloat16 multiply used by SVE2 instructions.

bits(N) BFMul(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean fpexc = TRUE;
    return BFMul(op1, op2, fpcr, fpexc);

// BFMul()
// =====
// Non-widening BFloat16 multiply following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates op1 * op2.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFMul(bits(N) op1, bits(N) op2, FPCRTType fpcr, boolean fpexc)

    assert N == 16;
    FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    bits(2*N) op1_s = op1 : Zeros(N);
    bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);

    if !done then
        inf1 = (type1 == FPTType Infinity);
        inf2 = (type2 == FPTType Infinity);
        zero1 = (type1 == FPTType Zero);
        zero2 = (type2 == FPTType Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, 2*N);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, 2*N);
        else
            result = FPRoundBF(value1*value2, fpcr, rounding, fpexc, 2*N);

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```





```

// BFMulAdd()
// =====
// Non-widening BFloat16 fused multiply-add used by SVE2 instructions.

bits(N) BFMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean fpexc = TRUE;
    return BFMulAdd(addend, op1, op2, fpcr, fpexc);

// BFMulAdd()
// =====
// Non-widening BFloat16 fused multiply-add following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates addend + op1*op2 with a single rounding.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRTType fpcr, boolean fpexc)

    assert N == 16;
    FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    bits(2*N) addend_s = addend : Zeros(N);
    bits(2*N) op1_s = op1 : Zeros(N);
    bits(2*N) op2_s = op2 : Zeros(N);
    (typeA,signA,valueA) = FPUnpack(addend_s, fpcr, fpexc);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    inf1 = (type1 == FPTYPE\_Infinity);
    inf2 = (type2 == FPTYPE\_Infinity);
    zero1 = (type1 == FPTYPE\_Zero);
    zero2 = (type2 == FPTYPE\_Zero);

    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend_s, op1_s, op2_s, fpcr, fpexc);

    if !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPTYPE\_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPTYPE\_Infinity);
        zeroA = (typeA == FPTYPE\_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', 2*N);
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', 2*N);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA, 2*N);

```

```

    // Otherwise calculate numerical result and round it.
else
    result_value = valueA + (value1 * value2);
    if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
        result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
        result = FPZero(result_sign, 2*N);
    else
        result = FPRoundBF(result_value, fpcr, rounding, fpexc, 2*N);

    if !invalidop && fpexc then
        FPProcessDenorms3(typeA, type1, type2, 2*N, fpcr);

return result<2*N-1:N>;

```

### Library pseudocode for shared/functions/float/bfloat/BFMulAddH

```

// BFMulAddH()
// =====
// Used by BFMLALB, BFMLALT, BFMLSLB and BFMLSLT instructions.

bits(N) BFMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRTType fpcr_in)
    assert N == 32;
    bits(N) value1 = op1 : Zeros(N DIV 2);
    bits(N) value2 = op2 : Zeros(N DIV 2);
    FPCRTType fpcr = fpcr_in;
    boolean altfp = HaveAltFP() && fpcr.AH == '1'; // When TRUE:
    boolean fpexc = !altfp; // Do not generate floating point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00'; // Use RNE rounding mode
    return FPMulAdd(addend, value1, value2, fpcr, fpexc);

```

### Library pseudocode for shared/functions/float/bfloat/BFMulAddH\_ZA

```

// BFMulAddH_ZA()
// =====
// Used by SME2 ZA-targeting BFMLAL and BFMLSL instructions.

bits(N) BFMulAddH_ZA(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRTType fpcr)
    assert N == 32;
    bits(N) value1 = op1 : Zeros(N DIV 2);
    bits(N) value2 = op2 : Zeros(N DIV 2);
    return FPMulAdd\_ZA(addend, value1, value2, fpcr);

```

### Library pseudocode for shared/functions/float/bfloat/BFMulAdd\_ZA

```

// BFMulAdd_ZA()
// =====
// Non-widening BFloat16 fused multiply-add used by SME2 ZA-targeting instructions.

bits(N) BFMulAdd_ZA(bits(N) addend, bits(N) op1, bits(N) op2, FPCRTType fpcr_in)
    boolean fpexc = FALSE;
    FPCRTType fpcr = fpcr_in;
    fpcr.DN = '1'; // Generate default NaN values
    return BFMulAdd(addend, op1, op2, fpcr, fpexc);

```

## Library pseudocode for shared/functions/float/bfloat/BFMulH

```
// BFMulH()
// =====
// BFloat16 widening multiply to single-precision following BFloat16
// computation behaviors.

bits(2*N) BFMulH(bits(N) op1, bits(N) op2)

    assert N == 16;
    bits(2*N) result;

    FPCRTYPE fpcr = FPCR[];
    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPTYPE\_QNaN || type2 == FPTYPE\_QNaN then
        result = FPDefaultNaN(fpcr, 2*N);
    else
        inf1 = (type1 == FPTYPE\_Infinity);
        inf2 = (type2 == FPTYPE\_Infinity);
        zero1 = (type1 == FPTYPE\_Zero);
        zero2 = (type2 == FPTYPE\_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr, 2*N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, 2*N);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, 2*N);
        else
            result = BFRound(value1*value2, 2*N);

    return result;
```

## Library pseudocode for shared/functions/float/bfloat/BFNeg

```
// BFNeg()
// =====

bits(N) BFNeg(bits(N) op)
    assert N == 16;
    boolean honor_altp = TRUE;    // Honor alternate handling
    return BFNeg(op, honor_altp);

// BFNeg()
// =====

bits(N) BFNeg(bits(N) op, boolean honor_altp)

    assert N == 16;
    if honor_altp && !UsingAArch32() && HaveAltFP() then
        FPCRTYPE fpcr = FPCR[];
        if fpcr.AH == '1' then
            boolean fpexc = FALSE;
            boolean isbfloat16 = TRUE;
            (fptype, -, -) = FPUnpackBase(op, fpcr, fpexc, isbfloat16);
            if fptype IN {FPTYPE\_SNaN, FPTYPE\_QNaN} then

                return op;    // When fpcr.AH=1, sign of NaN has no consequence

    return NOT(op<N-1>) : op<N-2:0>;
```

## Library pseudocode for shared/functions/float/bfloat/BFRound

```
// BFRound()
// =====
// Converts a real number OP into a single-precision value using the
// Round to Odd rounding mode and following BFloat16 computation behaviors.

bits(N) BFRound(real op, integer N)

    assert N == 32;
    assert op != 0.0;
    bits(N) result;

    // Format parameters - minimum exponent, numbers of exponent and fraction bits.
    minimum_exp = -126; E = 8; F = 23;

    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    real mantissa;
    if op < 0.0 then
        sign = '1'; mantissa = -op;
    else
        sign = '0'; mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0; exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0; exponent = exponent + 1;

    // Fixed Flush-to-zero.
    if exponent < minimum_exp then
        return FPZero(sign, N);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max((exponent - minimum_exp) + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F); // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Round to Odd
    if error != 0.0 then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if biased_exp >= 2^E - 1 then
        result = FPInfinity(sign, N); // Overflows generate appropriately-signed Infinity
    else
        result = sign : biased_exp<(N-2)-F:0> : int_mant<F-1:0>;

    return result;
```

## Library pseudocode for shared/functions/float/bfloat/BFSub

```
// BFSub()
// =====
// Non-widening BFloat16 subtraction used by SVE2 instructions.

bits(N) BFSub(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean fpexc = TRUE;
    return BFSub(op1, op2, fpcr, fpexc);

// BFSub()
// =====
// Non-widening BFloat16 subtraction following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates op1 - op2.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFSub(bits(N) op1, bits(N) op2, FPCRTType fpcr, boolean fpexc)

    assert N == 16;
    FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    bits(2*N) op1_s = op1 : Zeros(N);
    bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);

    if !done then
        inf1 = (type1 == FPTType Infinity);
        inf2 = (type2 == FPTType Infinity);
        zero1 = (type1 == FPTType Zero);
        zero2 = (type2 == FPTType Zero);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPExc InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', 2*N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', 2*N);
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1, 2*N);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding NEGINF then '1' else '0';
                result = FPZero(result_sign, 2*N);
            else
                result = FPRoundBF(result_value, fpcr, rounding, fpexc, 2*N);

            if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```

## Library pseudocode for shared/functions/float/bfloat/BFSub\_ZA

```
// BFSub_ZA()
// =====
// Non-widening BFloat16 subtraction used by SME2 ZA-targeting instructions.

bits(N) BFSub_ZA(bits(N) op1, bits(N) op2, FPCRTType fpcr_in)
    boolean fpexc = FALSE;
    FPCRTType fpcr = fpcr_in;
    fpcr.DN = '1'; // Generate default NaN values
    return BFSub(op1, op2, fpcr, fpexc);
```

## Library pseudocode for shared/functions/float/bfloat/BFUnpack

```
// BFUnpack()
// =====
// Unpacks a BFloat16 or single-precision value into its type,
// sign bit and real number that it represents.
// The real number result has the correct sign for numbers and infinities,
// is very large in magnitude for infinities, and is 0.0 for NaNs.
// (These values are chosen to simplify the description of
// comparisons and conversions.)

(FPType, bit, real) BFUnpack(bits(N) fpval)

    assert N IN {16,32};

    bit sign;
    bits(8) exp;
    bits(23) frac;
    if N == 16 then
        sign = fpval<15>;
        exp = fpval<14:7>;
        frac = fpval<6:0> : Zeros(16);
    else // N == 32
        sign = fpval<31>;
        exp = fpval<30:23>;
        frac = fpval<22:0>;

    FPType fptype;
    real value;
    if IsZero(exp) then
        fptype = FPType Zero; value = 0.0; // Fixed Flush to Zero
    elseif IsOnes(exp) then
        if IsZero(frac) then
            fptype = FPType Infinity; value = 2.0^1000000;
        else // no SNaN for BF16 arithmetic
            fptype = FPType QNaN; value = 0.0;
    else
        fptype = FPType Nonzero;
        value = 2.0^(UInt(exp)-127) * (1.0 + Real(UInt(frac)) * 2.0^-23);

    if sign == '1' then value = -value;

    return (fptype, sign, value);
```

## Library pseudocode for shared/functions/float/bfloat/BFZero

```
// BFZero()
// =====

bits(N) BFZero(bit sign, integer N)
    assert N == 16;
    constant integer E = 8;
    constant integer F = N - (E + 1);
    return sign : Zeros(E) : Zeros(F);
```

## Library pseudocode for shared/functions/float/bfloat/FAdd\_BF16

```
// FAdd_BF16()
// =====
// Single-precision add following BFloat16 computation behaviors.

bits(N) FAdd_BF16(bits(N) op1, bits(N) op2)

    assert N == 32;
    bits(N) result;

    FPCRTType fpcr = FPCR[];
    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPType\_QNaN || type2 == FPType\_QNaN then
        result = FPDefaultNaN(fpcr, N);
    else
        inf1 = (type1 == FPType\_Infinity);
        inf2 = (type2 == FPType\_Infinity);
        zero1 = (type1 == FPType\_Zero);
        zero2 = (type2 == FPType\_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr, N);
        elseif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0', N);
        elseif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1', N);
        elseif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, N);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then
                result = FPZero('0', N);    // Positive sign when Round to Odd
            else
                result = BFRound(result_value, N);

    return result;
```

## Library pseudocode for shared/functions/float/bfloat/FPConvertBF

```
// FPConvertBF()
// =====
// Converts a single-precision OP to BFloat16 value using the
// Round to Nearest Even rounding mode when executed from AArch64 state and
// FPCR.AH == '1', otherwise rounding is controlled by FPCR/FPSCR.

bits(N DIV 2) FPConvertBF(bits(N) op, FPCRTType fpcr_in, FPRounding rounding_in)

    assert N == 32;
    constant integer halfsize = N DIV 2;
    FPCRTType fpcr = fpcr_in;
    FPRounding rounding = rounding_in;
    bits(N) result; // BF16 value in top 16 bits
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean fpexc = !altfp; // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
    if altfp then rounding = FPRounding\_TIEEVEN; // Use RNE rounding mode

    // Unpack floating-point operand, with always flush-to-zero if fpcr.AH == '1'.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN then
        if fpcr.DN == '1' then
            result = FPDefaultNaN(fpcr, N);
        else
            result = FPConvertNaN(op, N);
        if fptype == FPTType\_SNaN then
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
    elseif fptype == FPTType\_Infinity then
        result = FPInfinity(sign, N);
    elseif fptype == FPTType\_Zero then
        result = FPZero(sign, N);
    else
        result = FPRoundBF(value, fpcr, rounding, fpexc, N);

    // Returns correctly rounded BF16 value from top 16 bits
    return result<(2*halfsize)-1:halfsize>;

// FPConvertBF()
// =====
// Converts a single-precision operand to BFloat16 value.

bits(N DIV 2) FPConvertBF(bits(N) op, FPCRTType fpcr)
    return FPConvertBF(op, fpcr, FPRoundingMode(fpcr));
```

## Library pseudocode for shared/functions/float/bfloat/FPRoundBF

```
// FPRoundBF()
// =====
// Converts a real number OP into a BFloat16 value using the supplied
// rounding mode RMODE. The 'fpexc' argument controls the generation of
// floating-point exceptions.

bits(N) FPRoundBF(real op, FPCRTType fpcr, FPRounding rounding, boolean fpexc, integer N)
    assert N == 32;
    boolean isbfloat16 = TRUE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);
```



## Library pseudocode for shared/functions/float/fixedtofp/FixedToFP

```
// FixedToFP()
// =====

// Convert M-bit fixed point 'op' with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCRTType fpcr,
FPRounding rounding, integer N)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(N) result;
    assert fbits >= 0;
    assert rounding != FPRounding\_ODD;

    // Correct signed-ness
    int_operand = Int(op, unsigned);

    // Scale by fractional bits and generate a real value
    real_operand = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = FPZero('0', N);
    else
        result = FPRound(real_operand, fpcr, rounding, N);

    return result;
```

## Library pseudocode for shared/functions/float/fpabs/FPAbs

```
// FPAbs()
// =====

bits(N) FPAbs(bits(N) op)

    assert N IN {16,32,64};
    if !UsingAArch32() && HaveAltFP() then
        FPCRTType fpcr = FPCR[];
        if fpcr.AH == '1' then
            (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
            if fptype IN {FPTType\_SNaN, FPTType\_QNaN} then
                return op; // When fpcr.AH=1, sign of NaN has no consequence

    return '0' : op<N-2:0>;
```

## Library pseudocode for shared/functions/float/fpadd/FPAdd

```
// FPAdd()
// =====

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean fpexc = TRUE;          // Generate floating-point exceptions
    return FPAdd(op1, op2, fpcr, fpexc);

// FPAdd()
// =====

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRTType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        inf1 = (type1 == FPTType\_Infinity);  inf2 = (type2 == FPTType\_Infinity);
        zero1 = (type1 == FPTType\_Zero);      zero2 = (type2 == FPTType\_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, N);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

        if fpexc then FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fpadd/FPAdd\_ZA

```
// FPAdd_ZA()
// =====
// Calculates op1+op2 for SME2 ZA-targeting instructions.

bits(N) FPAdd_ZA(bits(N) op1, bits(N) op2, FPCRTType fpcr_in)
    FPCRTType fpcr = fpcr_in;
    boolean fpexc = FALSE;          // Do not generate floating-point exceptions
    fpcr.DN = '1';                  // Generate default NaN values
    return FPAdd(op1, op2, fpcr, fpexc);
```

## Library pseudocode for shared/functions/float/fpcmpare/FPCompare

```
// FPCompare()
// =====

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRTType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    bits(4) result;
    if type1 IN {FPTType\_SNaN, FPTType\_QNaN} || type2 IN {FPTType\_SNaN, FPTType\_QNaN} then
        result = '0011';
        if type1 == FPTType\_SNaN || type2 == FPTType\_SNaN || signal_nans then
            FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = '0110';
        elseif value1 < value2 then
            result = '1000';
        else // value1 > value2
            result = '0010';

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fpcmpareeq/FPCompareEQ

```
// FPCompareEQ()
// =====

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRTType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    boolean result;
    if type1 IN {FPTType\_SNaN, FPTType\_QNaN} || type2 IN {FPTType\_SNaN, FPTType\_QNaN} then
        result = FALSE;
        if type1 == FPTType\_SNaN || type2 == FPTType\_SNaN then
            FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 == value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fpcomparege/FPCompareGE

```
// FPCompareGE()
// =====

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRTType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    boolean result;
    if type1 IN {FPTYPE\_SNaN, FPTYPE\_QNaN} || type2 IN {FPTYPE\_SNaN, FPTYPE\_QNaN} then
        result = FALSE;
        FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 >= value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fpcomparegt/FPCompareGT

```
// FPCompareGT()
// =====

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRTType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    boolean result;
    if type1 IN {FPTYPE\_SNaN, FPTYPE\_QNaN} || type2 IN {FPTYPE\_SNaN, FPTYPE\_QNaN} then
        result = FALSE;
        FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 > value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fpconvert/FPConvert

```
// FPConvert()
// =====

// Convert floating point 'op' with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.

bits(M) FPConvert(bits(N) op, FPCRTType fpcr, FPRounding rounding, integer M)

    assert M IN {16,32,64};
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (fptype,sign,value) = FPUnpackCV(op, fpcr);

    alt_hp = (M == 16) && (fpcr.AHP == '1');

    if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN then
        if alt_hp then
            result = FPZero(sign, M);
        elsif fpcr.DN == '1' then
            result = FPDefaultNaN(fpcr, M);
        else
            result = FPConvertNaN(op, M);
            if fptype == FPTType\_SNaN || alt_hp then
                FPProcessException(FPExc\_InvalidOp, fpcr);
            elsif fptype == FPTType\_Infinity then
                if alt_hp then
                    result = sign:Ones(M-1);
                    FPProcessException(FPExc\_InvalidOp, fpcr);
                else
                    result = FPInfinity(sign, M);
            elsif fptype == FPTType\_Zero then
                result = FPZero(sign, M);
            else
                result = FPRoundCV(value, fpcr, rounding, M);
                FPProcessDenorm(fptype, N, fpcr);

    return result;

// FPConvert()
// =====

bits(M) FPConvert(bits(N) op, FPCRTType fpcr, integer M)
    return FPConvert(op, fpcr, FPRoundingMode(fpcr), M);
```

## Library pseudocode for shared/functions/float/fpconvertnan/FPConvertNaN

```
// FPConvertNaN()
// =====
// Converts a NaN of one floating-point type to another

bits(M) FPConvertNaN(bits(N) op, integer M)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(M) result;
    bits(51) frac;

    sign = op<N-1>;

    // Unpack payload from input NaN
    case N of
        when 64 frac = op<50:0>;
        when 32 frac = op<21:0>:Zeros(29);
        when 16 frac = op<8:0>:Zeros(42);

    // Repack payload into output NaN, while
    // converting an SNaN to a QNaN.
    case M of
        when 64 result = sign:Ones(M-52):frac;
        when 32 result = sign:Ones(M-23):frac<50:29>;
        when 16 result = sign:Ones(M-10):frac<50:42>;

    return result;
```

## Library pseudocode for shared/functions/float/fpcrtype/FPCRTType

```
type FPCRTType;
```

## Library pseudocode for shared/functions/float/fpdecoderm/FPDecodeRM

```
// FPDecodeRM()
// =====
// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)

    FPRounding result;
    case rm of
        when '00' result = FPRounding_TIEAWAY; // A
        when '01' result = FPRounding_TIEEVEN; // N
        when '10' result = FPRounding_POSINF; // P
        when '11' result = FPRounding_NEGINF; // M

    return result;
```

## Library pseudocode for shared/functions/float/fpdecoderounding/FPDecodeRounding

```
// FPDecodeRounding()
// =====
// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)

    case rmode of
        when '00' return FPRounding_TIEEVEN; // N
        when '01' return FPRounding_POSINF; // P
        when '10' return FPRounding_NEGINF; // M
        when '11' return FPRounding_ZERO; // Z
```

## Library pseudocode for shared/functions/float/fpdefaultnan/FPDefaultNaN

```
// FPDefaultNaN()
// =====

bits(N) FPDefaultNaN(integer N)
  FPCRTType fpcr = FPCR\[\];
  return FPDefaultNaN(fpcr, N);

bits(N) FPDefaultNaN(FPCRTType fpcr, integer N)

  assert N IN {16,32,64};
  constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
  constant integer F = N - (E + 1);
  bit sign = if HaveAltFP() && !UsingAArch32() then fpcr.AH else '0';

  bits(E) exp = Ones(E);
  bits(F) frac = '1':Zeros(F-1);

  return sign : exp : frac;
```

## Library pseudocode for shared/functions/float/fpdiv/FPDiv

```
// FPDiv()
// =====

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCRTType fpcr)

  assert N IN {16,32,64};
  (type1,sign1,value1) = FPUnpack(op1, fpcr);
  (type2,sign2,value2) = FPUnpack(op2, fpcr);
  (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);

  if !done then
    inf1 = type1 == FPTType Infinity;
    inf2 = type2 == FPTType Infinity;
    zero1 = type1 == FPTType Zero;
    zero2 = type2 == FPTType Zero;

    if (inf1 && inf2) || (zero1 && zero2) then
      result = FPDefaultNaN(fpcr, N);
      FPProcessException(FPExc\_InvalidOp, fpcr);
    elsif inf1 || zero2 then
      result = FPInfinity(sign1 EOR sign2, N);
      if !inf1 then FPProcessException(FPExc\_DivideByZero, fpcr);
    elsif zero1 || inf2 then
      result = FPZero(sign1 EOR sign2, N);
    else
      result = FPRound(value1/value2, fpcr, N);

    if !zero2 then
      FPProcessDenorms(type1, type2, N, fpcr);

  return result;
```





```

// FPDot()
// =====
// Calculates single-precision result of 2-way 16-bit floating-point dot-product
// with a single rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'isbfloat16'
// determines whether input operands are BFloat16 or half-precision type.
// and 'fpexc' controls the generation of floating-point exceptions.

bits(N) FPDot(bits(N DIV 2) op1_a, bits(N DIV 2) op1_b, bits(N DIV 2) op2_a,
              bits(N DIV 2) op2_b, FPCRTType fpcr, boolean isbfloat16, integer N)
    boolean fpexc = TRUE; // Generate floating-point exceptions
    return FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc, N);

bits(N) FPDot(bits(N DIV 2) op1_a, bits(N DIV 2) op1_b, bits(N DIV 2) op2_a,
              bits(N DIV 2) op2_b, FPCRTType fpcr_in, boolean isbfloat16, boolean fpexc, integer N)
    FPCRTType fpcr = fpcr_in;

    assert N == 32;
    bits(N) result;
    boolean done;
    fpcr.AHP = '0'; // Ignore alternative half-precision option
    rounding = FPRoundingMode(fpcr);

    (type1_a, sign1_a, value1_a) = FPUnpackBase(op1_a, fpcr, fpexc, isbfloat16);
    (type1_b, sign1_b, value1_b) = FPUnpackBase(op1_b, fpcr, fpexc, isbfloat16);
    (type2_a, sign2_a, value2_a) = FPUnpackBase(op2_a, fpcr, fpexc, isbfloat16);
    (type2_b, sign2_b, value2_b) = FPUnpackBase(op2_b, fpcr, fpexc, isbfloat16);

    inf1_a = (type1_a == FPTType Infinity); zero1_a = (type1_a == FPTType Zero);
    inf1_b = (type1_b == FPTType Infinity); zero1_b = (type1_b == FPTType Zero);
    inf2_a = (type2_a == FPTType Infinity); zero2_a = (type2_a == FPTType Zero);
    inf2_b = (type2_b == FPTType Infinity); zero2_b = (type2_b == FPTType Zero);

    (done, result) = FPProcessNaNs4(type1_a, type1_b, type2_a, type2_b,
                                    op1_a, op1_b, op2_a, op2_b, fpcr, fpexc, N);

    if (((inf1_a && zero2_a) || (zero1_a && inf2_a)) &&
        ((inf1_b && zero2_b) || (zero1_b && inf2_b))) then
        result = FPDefaultNaN(fpcr, N);
        if fpexc then FPProcessException(FPExc InvalidOp, fpcr);

    if !done then
        // Determine sign and type products will have if it does not cause an Invalid
        // Operation.
        signPa = sign1_a EOR sign2_a;
        signPb = sign1_b EOR sign2_b;
        infPa = inf1_a || inf2_a;
        infPb = inf1_b || inf2_b;
        zeroPa = zero1_a || zero2_a;
        zeroPb = zero1_b || zero2_b;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        invalidop = ((inf1_a && zero2_a) || (zero1_a && inf2_a) ||
                    (inf1_b && zero2_b) || (zero1_b && inf2_b) || (infPa && infPb && signPa != signPb));

        if invalidop then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infPa && signPa == '0') || (infPb && signPb == '0') then
            result = FPInfinity('0', N);
        elsif (infPa && signPa == '1') || (infPb && signPb == '1') then
            result = FPInfinity('1', N);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroPa && zeroPb && signPa == signPb then
            result = FPZero(signPa, N);

```

```

// Otherwise calculate fused sum of products and round it.
else
    result_value = (value1_a * value2_a) + (value1_b * value2_b);
    if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
        result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
        result = FPZero(result_sign, N);
    else
        result = FPRound(result_value, fpcr, rounding, fpexc, N);

return result;

```

### Library pseudocode for shared/functions/float/fpdot/FPDotAdd

```

// FPDotAdd()
// =====
// Half-precision 2-way dot-product and add to single-precision.

bits(N) FPDotAdd(bits(N) addend, bits(N DIV 2) op1_a, bits(N DIV 2) op1_b,
                bits(N DIV 2) op2_a, bits(N DIV 2) op2_b, FPCRTType fpcr)
    assert N == 32;

    bits(N) prod;
    boolean isbfloat16 = FALSE;
    boolean fpexc = TRUE; // Generate floating-point exceptions
    prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc, N);
    result = FPAdd(addend, prod, fpcr, fpexc);

return result;

```

### Library pseudocode for shared/functions/float/fpdot/FPDotAdd\_ZA

```

// FPDotAdd_ZA()
// =====
// Half-precision 2-way dot-product and add to single-precision
// for SME ZA-targeting instructions.

bits(N) FPDotAdd_ZA(bits(N) addend, bits(N DIV 2) op1_a, bits(N DIV 2) op1_b,
                   bits(N DIV 2) op2_a, bits(N DIV 2) op2_b, FPCRTType fpcr_in)
    FPCRTType fpcr = fpcr_in;
    assert N == 32;

    bits(N) prod;
    boolean isbfloat16 = FALSE;
    boolean fpexc = FALSE; // Do not generate floating-point exceptions
    fpcr.DN = '1'; // Generate default NaN values
    prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc, N);
    result = FPAdd(addend, prod, fpcr, fpexc);

return result;

```

### Library pseudocode for shared/functions/float/fpexc/FPExc

```

// FPExc
// =====

enumeration FPExc {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                  FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};

```

## Library pseudocode for shared/functions/float/fpinfinity/FPInfinity

```
// FPInfinity()
// =====

bits(N) FPInfinity(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    bits(E) exp = Ones(E);
    bits(F) frac = Zeros(F);

    return sign : exp : frac;
```

## Library pseudocode for shared/functions/float/fpmatmul/FPMatMulAdd

```
// FPMatMulAdd()
// =====
//
// Floating point matrix multiply and add to same precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 2] * op2[2, 2])

bits(N) FPMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, integer esize, FPCRTType fpcr)

    assert N == esize * 2 * 2;
    bits(N) result;
    bits(esize) prod0, prod1, sum;

    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, esize];
            prod0 = FPMul(Elem[op1, 2*i + 0, esize],
                        Elem[op2, 2*j + 0, esize], fpcr);
            prod1 = FPMul(Elem[op1, 2*i + 1, esize],
                        Elem[op2, 2*j + 1, esize], fpcr);
            sum = FPAdd(sum, FPAdd(prod0, prod1, fpcr), fpcr);
            Elem[result, 2*i + j, esize] = sum;

    return result;
```

## Library pseudocode for shared/functions/float/fpmax/FPMax

```
// FPMax()
// =====

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return FPMax(op1, op2, fpcr, altfp);

// FPMax()
// =====
// Compare two inputs and return the larger value after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behavior.

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRTType fpcr_in, boolean altfp)

    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    FPCRTType fpcr = fpcr_in;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if altfp && type1 == FPTYPE\_Zero && type2 == FPTYPE\_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return FPZero(sign2, N);
    elseif altfp && (type1 IN {FPTYPE\_SNaN, FPTYPE\_QNaN} || type2 IN {FPTYPE\_SNaN, FPTYPE\_QNaN}) then
        // Alternate handling of NaN inputs
        FPProcessException(FPExc\_InvalidOp, fpcr);
        return (if type2 == FPTYPE\_Zero then FPZero(sign2, N) else op2);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        FPTYPE fptype;
        bit sign;
        real value;
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPTYPE\_Infinity then
            result = FPInfinity(sign, N);
        elseif fptype == FPTYPE\_Zero then
            sign = sign1 AND sign2;           // Use most positive sign
            result = FPZero(sign, N);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';

            result = FPRound(value, fpcr, rounding, TRUE, N);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fpmaxnormal/FPMaxNormal

```
// FPMaxNormal()
// =====

bits(N) FPMaxNormal(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = Ones(E-1):'0';
    frac = Ones(F);

    return sign : exp : frac;
```

## Library pseudocode for shared/functions/float/fpmaxnum/FPMaxNum

```
// FPMaxNum()
// =====

bits(N) FPMaxNum(bits(N) op1_in, bits(N) op2_in, FPCRTType fpcr)

    assert N IN {16,32,64};
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    boolean type1_nan = type1 IN {FPTType QNaN, FPTType SNaN};
    boolean type2_nan = type2 IN {FPTType QNaN, FPTType SNaN};
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as -Infinity.
        if type1 == FPTType QNaN && type2 != FPTType QNaN then
            op1 = FPInfinity('1', N);
        elsif type1 != FPTType QNaN && type2 == FPTType QNaN then
            op2 = FPInfinity('1', N);

    altfmaxfmin = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMax(op1, op2, fpcr, altfmaxfmin);

    return result;
```

## Library pseudocode for shared/functions/float/fpmerge/IsMerging

```
// IsMerging()
// =====
// Returns TRUE if the output elements other than the lowest are taken from
// the destination register.

boolean IsMerging(FPCRTType fpcr)
    bit nep = if HaveSME() && PSTATE.SM == '1' && !IsFullA64Enabled() then '0' else fpcr.NEP;
    return HaveAltFP() && !UsingAArch32() && nep == '1';
```

## Library pseudocode for shared/functions/float/fpmin/FPMin

```
// FPMin()
// =====

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return FPMin(op1, op2, fpcr, altfp);

// FPMin()
// =====
// Compare two operands and return the smaller operand after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative behavior.

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRTType fpcr_in, boolean altfp)

    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    FPCRTType fpcr = fpcr_in;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if altfp && type1 == FPTType\_Zero && type2 == FPTType\_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return FPZero(sign2, N);
    elseif altfp && (type1 IN {FPTType\_SNaN, FPTType\_QNaN} || type2 IN {FPTType\_SNaN, FPTType\_QNaN}) then
        // Alternate handling of NaN inputs
        FPProcessException(FPExc\_InvalidOp, fpcr);
        return (if type2 == FPTType\_Zero then FPZero(sign2, N) else op2);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        FPTType fptype;
        bit sign;
        real value;
        FPRounding rounding;
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPTType\_Infinity then
            result = FPInfinity(sign, N);
        elseif fptype == FPTType\_Zero then
            sign = sign1 OR sign2; // Use most negative sign
            result = FPZero(sign, N);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';

            result = FPRound(value, fpcr, rounding, TRUE, N);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fpminnum/FPMinNum

```
// FPMinNum()
// =====

bits(N) FPMinNum(bits(N) op1_in, bits(N) op2_in, FPCRTType fpcr)

    assert N IN {16,32,64};
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    boolean type1_nan = type1 IN {FPTYPE\_QNaN, FPTYPE\_SNaN};
    boolean type2_nan = type2 IN {FPTYPE\_QNaN, FPTYPE\_SNaN};
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as +Infinity.
        if type1 == FPTYPE\_QNaN && type2 != FPTYPE\_QNaN then
            op1 = FPInfinity('0', N);
        elsif type1 != FPTYPE\_QNaN && type2 == FPTYPE\_QNaN then
            op2 = FPInfinity('0', N);

    altfmaxfmin = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMIn(op1, op2, fpcr, altfmaxfmin);

    return result;
```

## Library pseudocode for shared/functions/float/fpmul/FPMul

```
// FPMul()
// =====

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRTType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPTYPE\_Infinity);
        inf2 = (type2 == FPTYPE\_Infinity);
        zero1 = (type1 == FPTYPE\_Zero);
        zero2 = (type2 == FPTYPE\_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, N);
        else
            result = FPRound(value1*value2, fpcr, N);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```





```

// FPMulAdd()
// =====

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean fpexc = TRUE;          // Generate floating-point exceptions
    return FPMulAdd(addend, op1, op2, fpcr, fpexc);

// FPMulAdd()
// =====
//
// Calculates addend + op1*op2 with a single rounding. The 'fpcr' argument
// supplies the FPCR control bits, and 'fpexc' controls the generation of
// floating-point exceptions.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2,
    FPCRTType fpcr, boolean fpexc)

    assert N IN {16,32,64};

    (typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    rounding = FPRoundingMode(fpcr);
    inf1 = (type1 == FPTType\_Infinity); zero1 = (type1 == FPTType\_Zero);
    inf2 = (type2 == FPTType\_Infinity); zero2 = (type2 == FPTType\_Zero);

    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);

    if !(HaveAltFP() && UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPTType\_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPTType\_Infinity); zeroA = (typeA == FPTType\_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', N);
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', N);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA, N);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

    if !invalidop && fpexc then

```

```

FPProcessDenorms3(typeA, type1, type2, N, fpcr);

return result;

```

### Library pseudocode for shared/functions/float/fpmuladd/FPMulAdd\_ZA

```

// FPMulAdd_ZA()
// =====
// Calculates addend + op1*op2 with a single rounding for SME ZA-targeting
// instructions.

bits(N) FPMulAdd_ZA(bits(N) addend, bits(N) op1, bits(N) op2, FPCRTType fpcr_in)
FPCRTType fpcr = fpcr_in;
boolean fpexc = FALSE;           // Do not generate floating-point exceptions
fpcr.DN = '1';                   // Generate default NaN values
return FPMulAdd(addend, op1, op2, fpcr, fpexc);

```



```

// FPMulAddH()
// =====
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRTType fpcr)
    boolean fpexc = TRUE;          // Generate floating-point exceptions
    return FPMulAddH(addend, op1, op2, fpcr, fpexc);

// FPMulAddH()
// =====
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2,
    FPCRTType fpcr, boolean fpexc)

    assert N == 32;
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    inf1 = (type1 == FPType\_Infinity); zero1 = (type1 == FPType\_Zero);
    inf2 = (type2 == FPType\_Infinity); zero2 = (type2 == FPType\_Zero);

    (done,result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);

    if !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPType\_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType\_Infinity); zeroA = (typeA == FPType\_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', N);
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', N);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA, N);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

    if !invalidop && fpexc then
        FPProcessDenorm(typeA, N, fpcr);

```

```
return result;
```

## Library pseudocode for shared/functions/float/fpmuladdh/FPMulAddH\_ZA

```
// FPMulAddH_ZA()
// =====
// Calculates addend + op1*op2 for SME2 ZA-targeting instructions.

bits(N) FPMulAddH_ZA(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRTType fpcr_in)
FPCRTType fpcr = fpcr_in;
boolean fpexc = FALSE;          // Do not generate floating-point exceptions
fpcr.DN = '1';                  // Generate default NaN values
return FPMulAddH(addend, op1, op2, fpcr, fpexc);
```

## Library pseudocode for shared/functions/float/fpmuladdh/FPPProcessNaNs3H

```
// FPPProcessNaNs3H()
// =====

(boolean, bits(N)) FPPProcessNaNs3H(FPTType type1, FPTType type2, FPTType type3,
bits(N) op1, bits(N DIV 2) op2, bits(N DIV 2) op3,
FPCRTType fpcr, boolean fpexc)

assert N IN {32,64};

bits(N) result;
FPTType type_nan;
// When TRUE, use alternative NaN propagation rules.
boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
boolean op1_nan = type1 IN {FPTType\_SNaN, FPTType\_QNaN};
boolean op2_nan = type2 IN {FPTType\_SNaN, FPTType\_QNaN};
boolean op3_nan = type3 IN {FPTType\_SNaN, FPTType\_QNaN};
if altfp then
    if (type1 == FPTType\_SNaN || type2 == FPTType\_SNaN || type3 == FPTType\_SNaN) then
        type_nan = FPTType\_SNaN;
    else
        type_nan = FPTType\_QNaN;

boolean done;
if altfp && op1_nan && op2_nan && op3_nan then // <n> register NaN selected
    done = TRUE; result = FPConvertNaN(FPPProcessNaN(type_nan, op2, fpcr, fpexc), N);
elseif altfp && op2_nan && (op1_nan || op3_nan) then // <n> register NaN selected
    done = TRUE; result = FPConvertNaN(FPPProcessNaN(type_nan, op2, fpcr, fpexc), N);
elseif altfp && op3_nan && op1_nan then // <m> register NaN selected
    done = TRUE; result = FPConvertNaN(FPPProcessNaN(type_nan, op3, fpcr, fpexc), N);
elseif type1 == FPTType\_SNaN then
    done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
elseif type2 == FPTType\_SNaN then
    done = TRUE; result = FPConvertNaN(FPPProcessNaN(type2, op2, fpcr, fpexc), N);
elseif type3 == FPTType\_SNaN then
    done = TRUE; result = FPConvertNaN(FPPProcessNaN(type3, op3, fpcr, fpexc), N);
elseif type1 == FPTType\_QNaN then
    done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
elseif type2 == FPTType\_QNaN then
    done = TRUE; result = FPConvertNaN(FPPProcessNaN(type2, op2, fpcr, fpexc), N);
elseif type3 == FPTType\_QNaN then
    done = TRUE; result = FPConvertNaN(FPPProcessNaN(type3, op3, fpcr, fpexc), N);
else
    done = FALSE; result = Zeros(N); // 'Don't care' result
return (done, result);
```

## Library pseudocode for shared/functions/float/fpmulx/FPMulX

```
// FPMulX()
// =====

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCRTType fpcr)

    assert N IN {16,32,64};
    bits(N) result;
    boolean done;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPTType Infinity);
        inf2 = (type2 == FPTType Infinity);
        zero1 = (type1 == FPTType Zero);
        zero2 = (type2 == FPTType Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo(sign1 EOR sign2, N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, N);
        else
            result = FPRound(value1*value2, fpcr, N);

            FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fpneg/FPNeg

```
// FPNeg()
// =====

bits(N) FPNeg(bits(N) op)

    assert N IN {16,32,64};
    if !UsingAArch32() && HaveAltFP() then
        FPCRTType fpcr = FPCR[];
        if fpcr.AH == '1' then
            (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
            if fptype IN {FPTType SNaN, FPType QNaN} then

                return op;          // When fpcr.AH=1, sign of NaN has no consequence

    return NOT(op<N-1>) : op<N-2:0>;
```

## Library pseudocode for shared/functions/float/fponepointfive/FPOnePointFive

```
// FPOnePointFive()
// =====

bits(N) FPOnePointFive(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-1);
    frac = '1':Zeros(F-1);
    result = sign : exp : frac;

    return result;
```

## Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorm

```
// FPProcessDenorm()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorm(FPTYPE fptype, integer N, FPCRTYPE fpcr)
    boolean altfp = HaveAltFP\(\) && !UsingAArch32\(\) && fpcr.AH == '1';
    if altfp && N != 16 && fptype == FPTYPE\_Denormal then
        FPProcessException(FPExc\_InputDenorm, fpcr);
```

## Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorms

```
// FPProcessDenorms()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms(FPTYPE type1, FPTYPE type2, integer N, FPCRTYPE fpcr)
    boolean altfp = HaveAltFP\(\) && !UsingAArch32\(\) && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPTYPE\_Denormal || type2 == FPTYPE\_Denormal) then
        FPProcessException(FPExc\_InputDenorm, fpcr);
```

## Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorms3

```
// FPProcessDenorms3()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms3(FPTYPE type1, FPTYPE type2, FPTYPE type3, integer N, FPCRTYPE fpcr)
    boolean altfp = HaveAltFP\(\) && !UsingAArch32\(\) && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPTYPE\_Denormal || type2 == FPTYPE\_Denormal ||
        type3 == FPTYPE\_Denormal) then
        FPProcessException(FPExc\_InputDenorm, fpcr);
```

## Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorms4

```
// FPProcessDenorms4()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms4(FPTYPE type1, FPTYPE type2, FPTYPE type3, FPTYPE type4, integer N, FPCRTYPE fpcr)
    boolean altfp = HaveAltFP\(\) && !UsingAArch32\(\) && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPTYPE\_Denormal || type2 == FPTYPE\_Denormal ||
        type3 == FPTYPE\_Denormal || type4 == FPTYPE\_Denormal) then
        FPProcessException(FPExc\_InputDenorm, fpcr);
```

## Library pseudocode for shared/functions/float/fpprocessexception/FPProcessException

```
// FPProcessException()
// =====
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc except, FPCRTYPE fpcr)

integer cumul;
// Determine the cumulative exception bit number
case except of
    when FPExc\_InvalidOp      cumul = 0;
    when FPExc\_DivideByZero    cumul = 1;
    when FPExc\_Overflow       cumul = 2;
    when FPExc\_Underflow      cumul = 3;
    when FPExc\_Inexact        cumul = 4;
    when FPExc\_InputDenorm    cumul = 7;
enable = cumul + 8;
if fpcr<enable> == '1' && (!HaveSME() || PSTATE.SM == '0' || IsFullA64Enabled()) then
    // Trapping of the exception enabled.
    // It is IMPLEMENTATION_DEFINED whether the enable bit may be set at all,
    // and if so then how exceptions and in what order that they may be
    // accumulated before calling FPTrappedException().
    bits(8) accumulated_exceptions = GetAccumulatedFPEExceptions();
    accumulated_exceptions<cumul> = '1';
    if boolean IMPLEMENTATION_DEFINED "Support trapping of floating-point exceptions" then
        if UsingAArch32() then
            AArch32.FPTrappedException(accumulated_exceptions);
        else
            is_ase = IsASEInstruction();
            AArch64.FPTrappedException(is_ase, accumulated_exceptions);
    else
        // The exceptions generated by this instruction are accumulated by the PE and
        // FPTrappedException is called later during its execution, before the next
        // instruction is executed. This field is cleared at the start of each FP instruction.
        SetAccumulatedFPEExceptions(accumulated_exceptions);
elseif UsingAArch32() then
    // Set the cumulative exception bit
    FPSCR<cumul> = '1';
else
    // Set the cumulative exception bit
    FPSR<cumul> = '1';

return;
```



## Library pseudocode for shared/functions/float/fpprocessnan/FPProcessNaN

```
// FPProcessNaN()
// =====

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCRTYPE fpcr)
    boolean fpexc = TRUE;    // Generate floating-point exceptions
    return FPProcessNaN(fptype, op, fpcr, fpexc);

// FPProcessNaN()
// =====
// Handle NaN input operands, returning the operand or default NaN value
// if fpcr.DN is selected. The 'fpcr' argument supplies the FPCR control bits.
// The 'fpexc' argument controls the generation of exceptions, regardless of
// whether 'fptype' is a signalling NaN or a quiet NaN.

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCRTYPE fpcr, boolean fpexc)

    assert N IN {16,32,64};
    assert fptype IN {FPType_QNaN, FPType_SNaN};
    integer topfrac;

    case N of
        when 16 topfrac = 9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;

    result = op;
    if fptype == FPType_SNaN then
        result<topfrac> = '1';
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    if fpcr.DN == '1' then // DefaultNaN requested
        result = FPDefaultNaN(fpcr, N);

    return result;
```

## Library pseudocode for shared/functions/float/fpprocessnans/FPProcessNaNs

```
// FPProcessNaNs()
// =====

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1,
                                bits(N) op2, FPCRTType fpcr)
    boolean fpexc      = TRUE;    // Generate floating-point exceptions
    return FPPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);

// FPProcessNaNs()
// =====
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'altfmaxfmin' controls
// alternative floating-point behavior for FMAX, FMIN and variants. 'fpexc'
// controls the generation of floating-point exceptions. Status information
// is updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1, bits(N) op2,
                                FPCRTType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    boolean altfp      = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean op1_nan    = type1 IN {FPType_SNaN, FPType_QNaN};
    boolean op2_nan    = type2 IN {FPType_SNaN, FPType_QNaN};
    boolean any_snan   = type1 == FPType_SNaN || type2 == FPType_SNaN;
    FPType type_nan    = if any_snan then FPType_SNaN else FPType_QNaN;

    if altfp && op1_nan && op2_nan then
        // <n> register NaN selected
        done = TRUE; result = FPPProcessNaN(type_nan, op1, fpcr, fpexc);
    elsif type1 == FPType_SNaN then
        done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPPProcessNaN(type2, op2, fpcr, fpexc);
    else
        done = FALSE; result = Zeros(N); // 'Don't care' result

    return (done, result);
```

## Library pseudocode for shared/functions/float/fpprocessnans3/FPPProcessNaNs3

```
// FPPProcessNaNs3()
// =====

(boolean, bits(N)) FPPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                   bits(N) op1, bits(N) op2, bits(N) op3,
                                   FPCRType fpcr)
    boolean fpexc = TRUE;    // Generate floating-point exceptions
    return FPPProcessNaNs3(type1, type2, type3, op1, op2, op3, fpcr, fpexc);

// FPPProcessNaNs3()
// =====
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

(boolean, bits(N)) FPPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                   bits(N) op1, bits(N) op2, bits(N) op3,
                                   FPCRType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    bits(N) result;
    boolean op1_nan = type1 IN {FPType SNaN, FPType QNaN};
    boolean op2_nan = type2 IN {FPType SNaN, FPType QNaN};
    boolean op3_nan = type3 IN {FPType SNaN, FPType QNaN};

    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    FPType type_nan;
    if altfp then
        if type1 == FPType SNaN || type2 == FPType SNaN || type3 == FPType SNaN then
            type_nan = FPType SNaN;
        else
            type_nan = FPType QNaN;

    boolean done;
    if altfp && op1_nan && op2_nan && op3_nan then
        // <n> register NaN selected
        done = TRUE; result = FPPProcessNaN(type_nan, op2, fpcr, fpexc);
    elsif altfp && op2_nan && (op1_nan || op3_nan) then
        // <n> register NaN selected
        done = TRUE; result = FPPProcessNaN(type_nan, op2, fpcr, fpexc);
    elsif altfp && op3_nan && op1_nan then
        // <m> register NaN selected
        done = TRUE; result = FPPProcessNaN(type_nan, op3, fpcr, fpexc);
    elsif type1 == FPType SNaN then
        done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType SNaN then
        done = TRUE; result = FPPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type3 == FPType SNaN then
        done = TRUE; result = FPPProcessNaN(type3, op3, fpcr, fpexc);
    elsif type1 == FPType QNaN then
        done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType QNaN then
        done = TRUE; result = FPPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type3 == FPType QNaN then
        done = TRUE; result = FPPProcessNaN(type3, op3, fpcr, fpexc);
    else
        done = FALSE; result = Zeros(N);    // 'Don't care' result

    return (done, result);
```

## Library pseudocode for shared/functions/float/fprocessnans4/FProcessNaNs4

```
// FProcessNaNs4()
// =====
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits.
// Status information is updated directly in the FPSR where appropriate.
// The 'fpexc' controls the generation of floating-point exceptions.

(boolean, bits(N)) FProcessNaNs4(FType type1, FType type2, FType type3, FType type4,
                                bits(N DIV 2) op1, bits(N DIV 2) op2, bits(N DIV 2) op3,
                                bits(N DIV 2) op4, FPCRTYPE fpcr, boolean fpexc, integer N)

    assert N == 32;

    bits(N) result;
    boolean done;
    // The FPCR.AH control does not affect these checks
    if type1 == FType\_SNaN then
        done = TRUE; result = FPConvertNaN(FProcessNaN(type1, op1, fpcr, fpexc), N);
    elsif type2 == FType\_SNaN then
        done = TRUE; result = FPConvertNaN(FProcessNaN(type2, op2, fpcr, fpexc), N);
    elsif type3 == FType\_SNaN then
        done = TRUE; result = FPConvertNaN(FProcessNaN(type3, op3, fpcr, fpexc), N);
    elsif type4 == FType\_SNaN then
        done = TRUE; result = FPConvertNaN(FProcessNaN(type4, op4, fpcr, fpexc), N);
    elsif type1 == FType\_QNaN then
        done = TRUE; result = FPConvertNaN(FProcessNaN(type1, op1, fpcr, fpexc), N);
    elsif type2 == FType\_QNaN then
        done = TRUE; result = FPConvertNaN(FProcessNaN(type2, op2, fpcr, fpexc), N);
    elsif type3 == FType\_QNaN then
        done = TRUE; result = FPConvertNaN(FProcessNaN(type3, op3, fpcr, fpexc), N);
    elsif type4 == FType\_QNaN then
        done = TRUE; result = FPConvertNaN(FProcessNaN(type4, op4, fpcr, fpexc), N);
    else
        done = FALSE; result = Zeros(N); // 'Don't care' result

    return (done, result);
```



```

// FPrecipEstimate()
// =====

bits(N) FPrecipEstimate(bits(N) operand, FPCRTType fpcr_in)

    assert N IN {16,32,64};
    FPCRTType fpcr = fpcr_in;
    bits(N) result;
    boolean overflow_to_inf;
    // When using alternative floating-point behavior, do not generate
    // floating-point exceptions, flush denormal input and output to zero,
    // and use RNE rounding mode.
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean fpexc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11';
    if altfp then fpcr.RMode = '00';

    (fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

    FPRounding rounding = FPRoundingMode(fpcr);
    if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN then
        result = FPProcessNaN(fptype, operand, fpcr, fpexc);
    elseif fptype == FPTType\_Infinity then
        result = FPZero(sign, N);
    elseif fptype == FPTType\_Zero then
        result = FPInfinity(sign, N);
        if fpexc then FPProcessException(FPExc\_DivideByZero, fpcr);
    elseif (
        (N == 16 && Abs(value) < 2.0^-16) ||
        (N == 32 && Abs(value) < 2.0^-128) ||
        (N == 64 && Abs(value) < 2.0^-1024)
    ) then
        case rounding of
            when FPRounding\_TIEEVEN
                overflow_to_inf = TRUE;
            when FPRounding\_POSINF
                overflow_to_inf = (sign == '0');
            when FPRounding\_NEGINF
                overflow_to_inf = (sign == '1');
            when FPRounding\_ZERO
                overflow_to_inf = FALSE;
        result = if overflow_to_inf then FPInfinity(sign, N) else FPMMaxNormal(sign, N);
        if fpexc then
            FPProcessException(FPExc\_Overflow, fpcr);
            FPProcessException(FPExc\_Inexact, fpcr);
    elseif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
        && (
            (N == 16 && Abs(value) >= 2.0^14) ||
            (N == 32 && Abs(value) >= 2.0^126) ||
            (N == 64 && Abs(value) >= 2.0^1022)
        ) then
        // Result flushed to zero of correct sign
        result = FPZero(sign, N);

        // Flush-to-zero never generates a trapped exception.
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            if fpexc then FPSR.UFC = '1';
    else
        // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
        // calculate result exponent. Scaled value has copied sign bit,
        // exponent = 1022 = double-precision biased version of -1,
        // fraction = original fraction
        bits(52) fraction;
        integer exp;
        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);

```

```

    when 32
        fraction = operand<22:0> : Zeros(29);
        exp = UInt(operand<30:23>);
    when 64
        fraction = operand<51:0>;
        exp = UInt(operand<62:52>);

if exp == 0 then
    if fraction<51> == '0' then
        exp = -1;
        fraction = fraction<49:0>:'00';
    else
        fraction = fraction<50:0>:'0';

integer scaled;
boolean increasedprecision = N==32 && HaveFeatRPRES() && altfp;

if !increasedprecision then
    scaled = UInt('1':fraction<51:44>);
else
    scaled = UInt('1':fraction<51:41>);

integer result_exp;
case N of
    when 16 result_exp = 29 - exp; // In range 29-30 = -1 to 29+1 = 30
    when 32 result_exp = 253 - exp; // In range 253-254 = -1 to 253+1 = 254
    when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

// Scaled is in range 256 .. 511 or 2048 .. 4095 range representing a
// fixed-point number in range [0.5 .. 1.0].
estimate = RecipEstimate(scaled, increasedprecision);

// Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
// fixed-point result in the range [1.0 .. 2.0].
// Convert to scaled floating point result with copied sign bit,
// high-order bits from estimate, and exponent calculated above.
if !increasedprecision then
    fraction = estimate<7:0> : Zeros(44);
else
    fraction = estimate<11:0> : Zeros(40);

if result_exp == 0 then
    fraction = '1' : fraction<51:1>;
elsif result_exp == -1 then
    fraction = '01' : fraction<51:2>;
    result_exp = 0;

case N of
    when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
    when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
    when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;

return result;

```

## Library pseudocode for shared/functions/float/fprecipestimate/RecipEstimate

```
// RecipEstimate()
// =====
// Compute estimate of reciprocal of 9-bit fixed-point number.
//
// a is in range 256 .. 511 or 2048 .. 4096 representing a number in
// the range  $0.5 \leq x < 1.0$ .
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191 representing a
// number in the range 1.0 to 511/256 or 1.00 to 8191/4096.

integer RecipEstimate(integer a_in, boolean increasedprecision)

    integer a = a_in;
    integer r;
    if !increasedprecision then
        assert 256 <= a && a < 512;
        a = a*2+1; // Round to nearest
        integer b = (2 ^ 19) DIV a;
        r = (b+1) DIV 2; // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 2048 <= a && a < 4096;
        a = a*2+1; // Round to nearest
        real real_val = Real(2^25)/Real(a);
        r = RoundDown(real_val);
        real error = real_val - Real(r);
        boolean round_up = error > 0.5; // Error cannot be exactly 0.5 so do not need tie case
        if round_up then r = r+1;
        assert 4096 <= r && r < 8192;

    return r;
```



## Library pseudocode for shared/functions/float/fprecpX/FPRecpX

```
// FPrecpX()
// =====

bits(N) FPrecpX(bits(N) op, FPCRTType fpcr_in)

    assert N IN {16,32,64};
    FPCRTType fpcr = fpcr_in;
    integer esize;
    case N of
        when 16 esize = 5;
        when 32 esize = 8;
        when 64 esize = 11;

    bits(N)                result;
    bits(esize)            exp;
    bits(esize)            max_exp;
    constant bits(N-(esize+1)) frac = Zeros(N-(esize+1));

    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    boolean fpexc = !altfp;           // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    case N of
        when 16 exp = op<(10+esize)-1:10>;
        when 32 exp = op<(23+esize)-1:23>;
        when 64 exp = op<(52+esize)-1:52>;

    max_exp = Ones(esize) - 1;

    if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN then
        result = FPProcessNaN(fptype, op, fpcr, fpexc);
    else
        if IsZero(exp) then           // Zero and denormals
            result = sign:max_exp:frac;
        else                          // Infinities and normals
            result = sign:NOT(exp):frac;

    return result;
```

## Library pseudocode for shared/functions/float/fpround/FPRound

```
// FPRound()
// =====
// Generic conversion from precise, unbounded real data type to IEEE format.

bits(N) FPRound(real op, FPCRTType fpcr, integer N)
    return FPRound(op, fpcr, FPRoundingMode(fpcr), N);

// FPRound()
// =====
// For directed FP conversion, includes an explicit 'rounding' argument.

bits(N) FPRound(real op, FPCRTType fpcr_in, FPRounding rounding, integer N)
    boolean fpexc = TRUE; // Generate floating-point exceptions
    return FPRound(op, fpcr_in, rounding, fpexc, N);

// FPRound()
// =====
// For AltFP, includes an explicit FPExc argument to disable exception
// generation and switches off Arm alternate half-precision mode.

bits(N) FPRound(real op, FPCRTType fpcr_in, FPRounding rounding, boolean fpexc, integer N)
    FPCRTType fpcr = fpcr_in;
    fpcr.AHP = '0';
    boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);
```



```

// FPRoundBase()
// =====
// For BFloat16, includes an explicit 'isbfloat16' argument.

bits(N) FPRoundBase(real op, FPCRTType fpcr, FPRounding rounding, boolean isbfloat16, integer N)
    boolean fpexc = TRUE;    // Generate floating-point exceptions
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);

// FPRoundBase()
// =====
// Convert a real number 'op' into an N-bit floating-point value using the
// supplied rounding mode 'rounding'.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

bits(N) FPRoundBase(real op, FPCRTType fpcr, FPRounding rounding,
    boolean isbfloat16, boolean fpexc, integer N)

    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding\_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    integer minimum_exp;
    integer F;
    integer E;
    if N == 16 then
        minimum_exp = -14;  E = 5;  F = 10;
    elsif N == 32 && isbfloat16 then
        minimum_exp = -126; E = 8;  F = 7;
    elsif N == 32 then
        minimum_exp = -126; E = 8;  F = 23;
    else // N == 64
        minimum_exp = -1022; E = 11; F = 52;

    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    real mantissa;
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // When TRUE, detection of underflow occurs after rounding and the test for a
    // denormalized number for single and double precision values occurs after rounding.
    altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

    // Deal with flush-to-zero before rounding if FPCR.AH != '1'.
    if (!altfp && ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) &&
        exponent < minimum_exp) then
        // Flush-to-zero never generates a trapped exception.
        if UsingAArch32() then
            FPSR.UFC = '1';
        else
            if fpexc then FPSR.UFC = '1';
        return FPZero(sign, N);

    biased_exp_unconstrained = (exponent - minimum_exp) + 1;
    int_mant_unconstrained = RoundDown(mantissa * 2.0^F);
    error_unconstrained = mantissa * 2.0^F - Real(int_mant_unconstrained);

    // Start creating the exponent value for the result. Start by biasing the actual exponent

```

```

// so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
biased_exp = Max((exponent - minimum_exp) + 1, 0);
if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

// Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
int_mant = RoundDown(mantissa * 2.0^F); // < 2.0^F if biased_exp == 0, >= 2.0^F if not
error = mantissa * 2.0^F - Real(int_mant);

// Underflow occurs if exponent is too small before rounding, and result is inexact or
// the Underflow exception is trapped. This applies before rounding if FPCR.AH != '1'.
boolean trapped_UF = fpcr.UFE == '1' && (!InStreamingMode() || IsFullA64Enabled());
if !altfp && biased_exp == 0 && (error != 0.0 || trapped_UF) then
    if fpexc then FPProcessException(FPExc Underflow, fpcr);

// Round result according to rounding mode.
boolean round_up_unconstrained;
boolean round_up;
boolean overflow_to_inf;
if altfp then

    case rounding of
        when FPRounding TIEEVEN
            round_up_unconstrained = (error_unconstrained > 0.5 ||
                (error_unconstrained == 0.5 && int_mant_unconstrained<0> == '1'));
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
            overflow_to_inf = TRUE;
        when FPRounding POSINF
            round_up_unconstrained = (error_unconstrained != 0.0 && sign == '0');
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding NEGINF
            round_up_unconstrained = (error_unconstrained != 0.0 && sign == '1');
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding ZERO, FPRounding ODD
            round_up_unconstrained = FALSE;
            round_up = FALSE;
            overflow_to_inf = FALSE;

    if round_up_unconstrained then
        int_mant_unconstrained = int_mant_unconstrained + 1;
        if int_mant_unconstrained == 2^(F+1) then // Rounded up to next exponent
            biased_exp_unconstrained = biased_exp_unconstrained + 1;
            int_mant_unconstrained = int_mant_unconstrained DIV 2;

// Deal with flush-to-zero and underflow after rounding if FPCR.AH == '1'.
if biased_exp_unconstrained < 1 && int_mant_unconstrained != 0 then
    // the result of unconstrained rounding is less than the minimum normalized number
    if (fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16) then // Flush-to-zero
        if fpexc then
            FPSR.UFC = '1';
            FPProcessException(FPExc Inexact, fpcr);
            return FPZero(sign, N);
        elsif error != 0.0 || trapped_UF then
            if fpexc then FPProcessException(FPExc Underflow, fpcr);
else // altfp == FALSE
    case rounding of
        when FPRounding TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
            overflow_to_inf = TRUE;
        when FPRounding POSINF
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding NEGINF
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding ZERO, FPRounding ODD
            round_up = FALSE;
            overflow_to_inf = FALSE;

```

```

if round_up then
    int_mant = int_mant + 1;
    if int_mant == 2^F then          // Rounded up from denormalized to normalized
        biased_exp = 1;
    if int_mant == 2^(F+1) then      // Rounded up to next exponent
        biased_exp = biased_exp + 1;
        int_mant = int_mant DIV 2;

// Handle rounding to odd
if error != 0.0 && rounding == FPRounding\_ODD then
    int_mant<0> = '1';

// Deal with overflow and generate result.
if N != 16 || fpcr.AHP == '0' then // Single, double or IEEE half precision
    if biased_exp >= 2^E - 1 then
        result = if overflow_to_inf then FPInfinity(sign, N) else FPMMaxNormal(sign, N);
        if fpexc then FPProcessException(FPExc\_Overflow, fpcr);
        error = 1.0; // Ensure that an Inexact exception occurs
    else
        result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));
else
    // Alternative half precision
    if biased_exp >= 2^E then
        result = sign : Ones(N-1);
        if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
        error = 0.0; // Ensure that an Inexact exception does not occur
    else
        result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));

// Deal with Inexact exception.
if error != 0.0 then
    if fpexc then FPProcessException(FPExc\_Inexact, fpcr);

return result;

```

### Library pseudocode for shared/functions/float/fpround/FPRoundCV

```

// FPRoundCV()
// =====
// Used for FP to FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

bits(N) FPRoundCV(real op, FPCRTType fpcr_in, FPRounding rounding, integer N)
    FPCRTType fpcr = fpcr_in;
    fpcr.FZ16 = '0';
    boolean fpexc = TRUE; // Generate floating-point exceptions
    boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);

```

### Library pseudocode for shared/functions/float/fprounding/FPRounding

```

// FPRounding
// =====
// The conversion and rounding functions take an explicit
// rounding mode enumeration instead of booleans or FPCR values.

enumeration FPRounding {FPRounding_TIEEVEN, FPRounding_POSINF,
                        FPRounding_NEGINF, FPRounding_ZERO,
                        FPRounding_TIEAWAY, FPRounding_ODD};

```

### Library pseudocode for shared/functions/float/fproundingmode/FPRoundingMode

```

// FPRoundingMode()
// =====
// Return the current floating-point rounding mode.

FPRounding FPRoundingMode(FPCRTType fpcr)
    return FPDecodeRounding(fpcr.RMode);

```

## Library pseudocode for shared/functions/float/fproundint/FPRoundInt

```
// FPRoundInt()
// =====

// Round op to nearest integral floating point value using rounding mode in FPCR/FPSCR.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to op.

bits(N) FPRoundInt(bits(N) op, FPCRTYPE fpcr, FPRounding rounding, boolean exact)

    assert rounding != FPRounding\_ODD;
    assert N IN {16,32,64};

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    bits(N) result;
    if fptype == FPTYPE\_SNaN || fptype == FPTYPE\_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPTYPE\_Infinity then
        result = FPInfinity(sign, N);
    elsif fptype == FPTYPE\_Zero then
        result = FPZero(sign, N);
    else
        // Extract integer component.
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        boolean round_up;
        case rounding of
            when FPRounding\_TIEEVEN
                round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
            when FPRounding\_POSINF
                round_up = (error != 0.0);
            when FPRounding\_NEGINF
                round_up = FALSE;
            when FPRounding\_ZERO
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding\_TIEAWAY
                round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

        if round_up then int_result = int_result + 1;

        // Convert integer value into an equivalent real value.
        real_result = Real(int_result);

        // Re-encode as a floating-point value, result is always exact.
        if real_result == 0.0 then
            result = FPZero(sign, N);
        else
            result = FPRound(real_result, fpcr, FPRounding\_ZERO, N);

        // Generate inexact exceptions.
        if error != 0.0 && exact then
            FPProcessException(FPExc\_Inexact, fpcr);

    return result;
```



```

// FPRoundIntN()
// =====

bits(N) FPRoundIntN(bits(N) op, FPCRType fpcr, FPRounding rounding, integer intsize)
    assert rounding != FPRounding\_ODD;
    assert N IN {32,64};
    assert intsize IN {32, 64};
    integer exp;
    bits(N) result;
    boolean round_up;
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - (E + 1);

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    if fptype IN {FPTType\_SNaN, FPType\_QNaN, FPTType\_Infinity} then
        if N == 32 then
            exp = 126 + intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
        else
            exp = 1022+intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
            FPProcessException(FPExc\_InvalidOp, fpcr);
    elseif fptype == FPTType\_Zero then
        result = FPZero(sign, N);
    else
        // Extract integer component.
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        case rounding of
            when FPRounding\_TIEEVEN
                round_up = error > 0.5 || (error == 0.5 && int_result<0> == '1');
            when FPRounding\_POSINF
                round_up = error != 0.0;
            when FPRounding\_NEGINF
                round_up = FALSE;
            when FPRounding\_ZERO
                round_up = error != 0.0 && int_result < 0;
            when FPRounding\_TIEAWAY
                round_up = error > 0.5 || (error == 0.5 && int_result >= 0);

        if round_up then int_result = int_result + 1;
        overflow = int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1);

        if overflow then
            if N == 32 then
                exp = 126 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
            else
                exp = 1022 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
                FPProcessException(FPExc\_InvalidOp, fpcr);
                // This case shouldn't set Inexact.
                error = 0.0;

        else
            // Convert integer value into an equivalent real value.
            real_result = Real(int_result);

            // Re-encode as a floating-point value, result is always exact.
            if real_result == 0.0 then
                result = FPZero(sign, N);

```



```
        else
            result = FPRound(real_result, fpcr, FPRounding\_ZERO, N);

// Generate inexact exceptions.
if error != 0.0 then
    FPProcessException(FPExc\_Inexact, fpcr);

return result;
```



```

// FPRSqrtEstimate()
// =====

bits(N) FPRSqrtEstimate(bits(N) operand, FPCRTType fpcr_in)

    assert N IN {16,32,64};
    FPCRTType fpcr = fpcr_in;

    // When using alternative floating-point behavior, do not generate
    // floating-point exceptions and flush denormal input to zero.
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean fpexc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11';

    (fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

    bits(N) result;
    if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN then
        result = FPProcessNaN(fptype, operand, fpcr, fpexc);
    elsif fptype == FPTType\_Zero then
        result = FPInfinity(sign, N);
        if fpexc then FPProcessException(FPExc\_DivideByZero, fpcr);
    elsif sign == '1' then
        result = FPDefaultNaN(fpcr, N);
        if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
    elsif fptype == FPTType\_Infinity then
        result = FPZero('0', N);
    else
        // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
        // evenness or oddness of the exponent unchanged, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
        // biased version of -1 or -2, fraction = original fraction extended with zeros.

        bits(52) fraction;
        integer exp;
        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            while fraction<51> == '0' do
                fraction = fraction<50:0> : '0';
                exp = exp - 1;
            fraction = fraction<50:0> : '0';

        integer scaled;
        boolean increasedprecision = N==32 && HaveFeatRPRES() && altfp;

        if !increasedprecision then
            if exp<0> == '0' then
                scaled = UInt('1':fraction<51:44>);
            else
                scaled = UInt('01':fraction<51:45>);
        else
            if exp<0> == '0' then
                scaled = UInt('1':fraction<51:41>);
            else
                scaled = UInt('01':fraction<51:42>);

        integer result_exp;
        case N of
            when 16 result_exp = ( 44 - exp) DIV 2;
            when 32 result_exp = ( 380 - exp) DIV 2;

```

```

    when 64 result_exp = (3068 - exp) DIV 2;

estimate = RecipSqrtEstimate(scaled, increasedprecision);

// Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
// fixed-point result in the range [1.0 .. 2.0].
// Convert to scaled floating point result with copied sign bit and high-order
// fraction bits, and exponent calculated above.
case N of
    when 16 result = '0' : result_exp<N-12:0> : estimate<7:0>:Zeros(2);
    when 32
        if !increasedprecision then
            result = '0' : result_exp<N-25:0> : estimate<7:0>:Zeros(15);
        else
            result = '0' : result_exp<N-25:0> : estimate<11:0>:Zeros(11);
    when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>:Zeros(44);

return result;

```

## Library pseudocode for shared/functions/float/fpsqrtestimate/RecipSqrtEstimate

```
// RecipSqrtEstimate()
// =====
// Compute estimate of reciprocal square root of 9-bit fixed-point number.
//
// a_in is in range 128 .. 511 or 1024 .. 4095, with increased precision,
// representing a number in the range 0.25 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191, with increased precision,
// representing a number in the range 1.0 to 511/256 or 8191/4096.

integer RecipSqrtEstimate(integer a_in, boolean increasedprecision)

    integer a = a_in;
    integer r;
    if !increasedprecision then
        assert 128 <= a && a < 512;
        if a < 256 then // 0.25 .. 0.5
            a = a*2+1; // a in units of 1/512 rounded to nearest
        else // 0.5 .. 1.0
            a = (a >> 1) << 1; // Discard bottom bit
            a = (a+1)*2; // a in units of 1/256 rounded to nearest
        integer b = 512;
        while a*(b+1)*(b+1) < 2^28 do
            b = b+1;
        // b = largest b such that b < 2^14 / sqrt(a)
        r = (b+1) DIV 2; // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 1024 <= a && a < 4096;
        real real_val;
        real error;
        integer int_val;

        if a < 2048 then // 0.25... 0.5
            a = a*2 + 1; // Take 10 bits of fraction and force a 1 at the bottom
            real_val = Real(a)/2.0;
        else // 0.5..1.0
            a = (a >> 1) << 1; // Discard bottom bit
            a = a+1; // Take 10 bits of fraction and force a 1 at the bottom
            real_val = Real(a);

        real_val = Sqrt(real_val); // This number will lie in the range of 32 to 64
        // Round to nearest even for a DP float number
        real_val = real_val * Real(2^47); // The integer is the size of the whole DP mantissa
        int_val = RoundDown(real_val); // Calculate rounding value
        error = real_val - Real(int_val);
        round_up = error > 0.5; // Error cannot be exactly 0.5 so do not need tie case
        if round_up then int_val = int_val+1;

        real_val = Real(2^65)/Real(int_val); // Lies in the range 4096 <= real_val < 8192
        int_val = RoundDown(real_val); // Round that (to nearest even) to give integer
        error = real_val - Real(int_val);
        round_up = (error > 0.5 || (error == 0.5 && int_val<0> == '1'));
        if round_up then int_val = int_val+1;

        r = int_val;
        assert 4096 <= r && r < 8192;

    return r;
```

## Library pseudocode for shared/functions/float/fpsqrt/FPSqrt

```
// FPSqrt()
// =====

bits(N) FPSqrt(bits(N) op, FPCRTType fpcr)

    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(op, fpcr);

    bits(N) result;
    if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPTType\_Zero then
        result = FPZero(sign, N);
    elsif fptype == FPTType\_Infinity && sign == '0' then
        result = FPInfinity(sign, N);
    elsif sign == '1' then
        result = FPDefaultNaN(fpcr, N);
        FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        result = FPRound(Sqrt(value), fpcr, N);
        FPProcessDenorm(fptype, N, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fpsub/FPSub

```
// FPSub()
// =====

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean fpexc = TRUE;          // Generate floating-point exceptions
    return FPSub(op1, op2, fpcr, fpexc);

// FPSub()
// =====

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRTType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        inf1 = (type1 == FPTType\_Infinity);
        inf2 = (type2 == FPTType\_Infinity);
        zero1 = (type1 == FPTType\_Zero);
        zero2 = (type2 == FPTType\_Zero);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1, N);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

            if fpexc then FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fpsub/FPSub\_ZA

```
// FPSub_ZA()
// =====
// Calculates op1-op2 for SME2 ZA-targeting instructions.

bits(N) FPSub_ZA(bits(N) op1, bits(N) op2, FPCRTType fpcr_in)
    FPCRTType fpcr = fpcr_in;
    boolean fpexc = FALSE;          // Do not generate floating-point exceptions
    fpcr.DN = '1';                  // Generate default NaN values
    return FPSub(op1, op2, fpcr, fpexc);
```

## Library pseudocode for shared/functions/float/fpthree/FPThree

```
// FPThree()
// =====

bits(N) FPThree(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '1':Zeros(E-1);
    frac = '1':Zeros(F-1);
    result = sign : exp : frac;

    return result;
```



## Library pseudocode for shared/functions/float/fptofixed/FPToFixed

```
// FPToFixed()
// =====

// Convert N-bit precision floating point 'op' to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRTYPE fpcr,
FPRounding rounding, integer M)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    assert fbits >= 0;
    assert rounding != FPRounding\_ODD;

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    // If NaN, set cumulative flag or take exception.
    if fptype == FPTYPE\_SNaN || fptype == FPTYPE\_QNaN then
        FPProcessException(FPEXC\_InvalidOp, fpcr);

    // Scale by fractional bits and produce integer rounded towards minus-infinity.
    value = value * 2.0^fbits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.
    boolean round_up;
    case rounding of
        when FPRounding\_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPRounding\_POSINF
            round_up = (error != 0.0);
        when FPRounding\_NEGINF
            round_up = FALSE;
        when FPRounding\_ZERO
            round_up = (error != 0.0 && int_result < 0);
        when FPRounding\_TIEAWAY
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

    if round_up then int_result = int_result + 1;

    // Generate saturated result and exceptions.
    (result, overflow) = SatQ(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPEXC\_InvalidOp, fpcr);
    elsif error != 0.0 then
        FPProcessException(FPEXC\_Inexact, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fptofixedjs/FPToFixedJS

```
// FPToFixedJS()
// =====

// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

(bits(N), bit) FPToFixedJS(bits(M) op, FPCRTType fpcr, boolean Is64, integer N)

    assert M == 64 && N == 32;

    // If FALSE, never generate Input Denormal floating-point exceptions.
    fpexc_idenorm = !(HaveAltFP() && UsingAArch32() && fpcr.AH == '1');

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc_idenorm);

    z = '1';
    // If NaN, set cumulative flag or take exception.
    if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN then
        FPProcessException(FPExc\_InvalidOp, fpcr);
        z = '0';

    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.

    round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1;

    integer result;
    if int_result < 0 then
        result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
    else
        result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));

    // Generate exceptions.
    if int_result < -(2^31) || int_result > (2^31)-1 then
        FPProcessException(FPExc\_InvalidOp, fpcr);
        z = '0';
    elsif error != 0.0 then
        FPProcessException(FPExc\_Inexact, fpcr);
        z = '0';
    elsif sign == '1' && value == 0.0 then
        z = '0';
    elsif sign == '0' && value == 0.0 && IsZero(op<51:0>) then
        z = '0';

    if fptype == FPTType\_Infinity then result = 0;

    return (result<N-1:0>, z);
```

## Library pseudocode for shared/functions/float/fptwo/FPTwo

```
// FPTwo()
// =====

bits(N) FPTwo(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '1':Zeros(E-1);
    frac = Zeros(F);
    result = sign : exp : frac;

    return result;
```

## Library pseudocode for shared/functions/float/fptype/FPType

```
// FPType
// =====

enumeration FPType {FPType_Zero,
                    FPType_Denormal,
                    FPType_Nonzero,
                    FPType_Infinity,
                    FPType_QNaN,
                    FPType_SNaN};
```

## Library pseudocode for shared/functions/float/fpunpack/FPUnpack

```
// FPUnpack()
// =====

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRTType fpcr_in)
    FPCRTType fpcr = fpcr_in;
    fpcr.AHP = '0';
    boolean fpexc = TRUE;    // Generate floating-point exceptions
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);

// FPUnpack()
// =====
//
// Used by data processing, int/fixed to FP and FP to int/fixed conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRTType fpcr_in, boolean fpexc)
    FPCRTType fpcr = fpcr_in;
    fpcr.AHP = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);
```



```

// FPUntpackBase()
// =====

(FPType, bit, real) FPUntpackBase(bits(N) fpval, FPCRTType fpcr, boolean fpexc)
    boolean isbfloat16 = FALSE;
    (fp_type, sign, value) = FPUntpackBase(fpval, fpcr, fpexc, isbfloat16);
    return (fp_type, sign, value);

// FPUntpackBase()
// =====
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr_in' argument supplies FPCR control bits, 'fpexc' controls the
// generation of floating-point exceptions and 'isbfloat16' determines whether
// N=16 signifies BFloat16 or half-precision type. Status information is updated
// directly in the FPSR where appropriate.

(FPType, bit, real) FPUntpackBase(bits(N) fpval, FPCRTType fpcr_in, boolean fpexc,
    boolean isbfloat16)

    assert N IN {16,32,64};

    FPCRTType fpcr = fpcr_in;

    boolean altfp = HaveAltFP() && !UsingAArch32();
    boolean fiz   = altfp && fpcr.FIZ == '1';
    boolean fz     = fpcr.FZ == '1' && !(altfp && fpcr.AH == '1');
    real value;
    bit sign;
    FPType fptype;

    if N == 16 && !isbfloat16 then
        sign = fpval<15>;
        exp16 = fpval<14:10>;
        frac16 = fpval<9:0>;
        if IsZero(exp16) then
            if IsZero(frac16) || fpcr.FZ16 == '1' then
                fptype = FPType\_Zero; value = 0.0;
            else
                fptype = FPType\_Denormal; value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
        elsif IsOnes(exp16) && fpcr.AHP == '0' then // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                fptype = FPType\_Infinity; value = 2.0^1000000;
            else
                fptype = if frac16<9> == '1' then FPType\_QNaN else FPType\_SNaN;
                value = 0.0;
        else
            fptype = FPType\_Nonzero;
            value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

    elsif N == 32 || isbfloat16 then
        bits(8) exp32;
        bits(23) frac32;
        if isbfloat16 then
            sign = fpval<15>;
            exp32 = fpval<14:7>;
            frac32 = fpval<6:0> : Zeros(16);
        else
            sign = fpval<31>;
            exp32 = fpval<30:23>;
            frac32 = fpval<22:0>;

        if IsZero(exp32) then
            if IsZero(frac32) then
                // Produce zero if value is zero.

```

```

    fptype = FPTYPE\_Zero; value = 0.0;
elseif fz || fiz then // Flush-to-zero if FIZ==1 or AH,FZ==01
    fptype = FPTYPE\_Zero; value = 0.0;
    // Check whether to raise Input Denormal floating-point exception.
    // fpcr.FIZ==1 does not raise Input Denormal exception.
    if fz then
        // Denormalized input flushed to zero
        if fpexc then FPPProcessException(FPExc\_InputDenorm, fpcr);
    else
        fptype = FPTYPE\_Denormal; value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
elseif IsOnes(exp32) then
    if IsZero(frac32) then
        fptype = FPTYPE\_Infinity; value = 2.0^1000000;
    else
        fptype = if frac32<22> == '1' then FPTYPE\_QNaN else FPTYPE\_SNaN;
        value = 0.0;
else
    fptype = FPTYPE\_Nonzero;
    value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);

else // N == 64
    sign = fpval<63>;
    exp64 = fpval<62:52>;
    frac64 = fpval<51:0>;

    if IsZero(exp64) then
        if IsZero(frac64) then
            // Produce zero if value is zero.
            fptype = FPTYPE\_Zero; value = 0.0;
        elseif fz || fiz then // Flush-to-zero if FIZ==1 or AH,FZ==01
            fptype = FPTYPE\_Zero; value = 0.0;
            // Check whether to raise Input Denormal floating-point exception.
            // fpcr.FIZ==1 does not raise Input Denormal exception.
            if fz then
                // Denormalized input flushed to zero
                if fpexc then FPPProcessException(FPExc\_InputDenorm, fpcr);
        else
            fptype = FPTYPE\_Denormal; value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
    elseif IsOnes(exp64) then
        if IsZero(frac64) then
            fptype = FPTYPE\_Infinity; value = 2.0^1000000;
        else
            fptype = if frac64<51> == '1' then FPTYPE\_QNaN else FPTYPE\_SNaN;
            value = 0.0;
    else
        fptype = FPTYPE\_Nonzero;
        value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);

    if sign == '1' then value = -value;

    return (fptype, sign, value);

```

## Library pseudocode for shared/functions/float/fpunpack/FPUnpackCV

```

// FPUnpackCV()
// =====
//
// Used for FP to FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

(FPType, bit, real) FPUnpackCV(bits(N) fpval, FPCRTType fpcr_in)
FPCRTType fpcr = fpcr_in;
fpcr.FZ16 = '0';
boolean fpexc = TRUE; // Generate floating-point exceptions
(fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
return (fp_type, sign, value);

```

## Library pseudocode for shared/functions/float/fpzero/FPZero

```
// FPZero()
// =====

bits(N) FPZero(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = Zeros(E);
    frac = Zeros(F);
    result = sign : exp : frac;

    return result;
```

## Library pseudocode for shared/functions/float/vfpexpandimm/VFPExpandImm

```
// VFPExpandImm()
// =====

bits(N) VFPExpandImm(bits(8) imm8, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = (N - E) - 1;
    sign = imm8<7>;
    exp = NOT(imm8<6>) : Replicate(imm8<6>,E-3) : imm8<5:4>;
    frac = imm8<3:0> : Zeros(F-4);
    result = sign : exp : frac;

    return result;
```

## Library pseudocode for shared/functions/integer/AddWithCarry

```
// AddWithCarry()
// =====
// Integer addition with carry input, returning result and NZCV flags

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    bit n = result<N-1>;
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

## Library pseudocode for shared/functions/interrupts/InterruptID

```
// InterruptID
// =====

enumeration InterruptID {
    InterruptID_PMIIRQ,
    InterruptID_COMMIRQ,
    InterruptID_CTIIRQ,
    InterruptID_COMMRX,
    InterruptID_COMMTX,
    InterruptID_CNTP,
    InterruptID_CNTHP,
    InterruptID_CNTHPS,
    InterruptID_CNTPS,
    InterruptID_CNTV,
    InterruptID_CNTHV,
    InterruptID_CNTHVS,
    InterruptID_PMBIRQ,
};
```

## Library pseudocode for shared/functions/interrupts/SetInterruptRequestLevel

```
// SetInterruptRequestLevel()
// =====
// Set a level-sensitive interrupt to the specified level.

SetInterruptRequestLevel(InterruptID id, Signal level);
```

## Library pseudocode for shared/functions/memory/AArch64.BranchAddr

```
// AArch64.BranchAddr()
// =====
// Return the virtual address with tag bits removed.
// This is typically used when the address will be stored to the program counter.

bits(64) AArch64.BranchAddr(bits(64) vaddress, bits(2) el)
    assert !UsingAArch32();
    msbit = AddrTop(vaddress, TRUE, el);
    if msbit == 63 then
        return vaddress;
    elsif (el IN {EL0, EL1} || IsInHost()) && vaddress<msbit> == '1' then
        return SignExtend(vaddress<msbit:0>, 64);
    else
        return ZeroExtend(vaddress<msbit:0>, 64);
```



## Library pseudocode for shared/functions/memory/AccessDescriptor

```
// AccessDescriptor
// =====
// Memory access or translation invocation details that steer architectural behavior

type AccessDescriptor is (
    AccessType acctype,
    bits(2) el, // Acting EL for the access
    SecurityState ss, // Acting Security State for the access
    boolean acqsc, // Acquire with Sequential Consistency
    boolean acqpc, // FEAT_LRCPC: Acquire with Processor Consistency
    boolean relsc, // Release with Sequential Consistency
    boolean limitedordered, // FEAT_LOR: Acquire/Release with limited ordering
    boolean exclusive, // Access has Exclusive semantics
    boolean atomicop, // FEAT_LSE: Atomic read-modify-write access
    MemAtomicOp modop, // FEAT_LSE: The modification operation in the 'atomicop' access
    boolean nontemporal, // Hints the access is non-temporal
    boolean read, // Read from memory or only require read permissions
    boolean write, // Write to memory or only require write permissions
    CacheOp cacheop, // DC/IC: Cache operation
    CacheOpScope opscope, // DC/IC: Scope of cache operation
    CacheType cachetype, // DC/IC: Type of target cache
    boolean pan, // FEAT_PAN: The access is subject to PSTATE.PAN
    boolean transactional, // FEAT_TME: Access is part of a transaction
    boolean nonfault, // SVE: Non-faulting load
    boolean firstfault, // SVE: First-fault load
    boolean first, // SVE: First-fault load for the first active element
    boolean contiguous, // SVE: Contiguous load/store not gather load/scatter store
    boolean streamingsve, // SME: Access made by PE while in streaming SVE mode
    boolean ls64, // FEAT_LS64: Accesses by accelerator support loads/stores
    boolean mops, // FEAT_MOPS: Memory operation (CPY/SET) accesses
    boolean rcw, // FEAT_THE: Read-Check-Write access
    boolean rcws, // FEAT_THE: Read-Check-Write Software access
    boolean toplevel, // FEAT_THE: Translation table walk access for TTB address
    VARange varange, // FEAT_THE: The corresponding TTBR supplying the TTB
    boolean a32lsmd, // A32 Load/Store Multiple Data access
    boolean tagchecked, // FEAT_MTE2: Access is tag checked
    boolean tagaccess, // FEAT_MTE: Access targets the tag bits
    boolean ispair, // Access represents a Load/Store pair access
    boolean highestaddressfirst, // FEAT_LRCPC3: Highest address is accessed first
    MPAMInfo mpam // FEAT_MPAM: MPAM information
)
```

## Library pseudocode for shared/functions/memory/AccessType

```
// AccessType
// =====

enumeration AccessType {
    AccessType_IFETCH, // Instruction FETCH
    AccessType_GPR, // Software load/store to a General Purpose Register
    AccessType_ASIMD, // Software ASIMD extension load/store instructions
    AccessType_SVE, // Software SVE load/store instructions
    AccessType_SME, // Software SME load/store instructions
    AccessType_IC, // Sysop IC
    AccessType_DC, // Sysop DC (not DC {Z,G,GZ}VA)
    AccessType_DCZero, // Sysop DC {Z,G,GZ}VA
    AccessType_AT, // Sysop AT
    AccessType_NV2, // NV2 memory redirected access
    AccessType_SPE, // Statistical Profiling buffer access
    AccessType_GCS, // Guarded Control Stack access
    AccessType_TRBE, // Trace Buffer access
    AccessType_GPTW, // Granule Protection Table Walk
    AccessType_TTW // Translation Table Walk
};
```

## Library pseudocode for shared/functions/memory/AddrTop

```
// AddrTop()
// =====
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

integer AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    if ELUsingAArch32(regime) then
        // AArch32 translation regime.
        return 31;
    else
        if EffectiveTBI(address, IsInstr, el) == '1' then
            return 55;
        else
            return 63;
```

## Library pseudocode for shared/functions/memory/AlignmentEnforced

```
// AlignmentEnforced()
// =====
// For the active translation regime, determine if alignment is required by all accesses

boolean AlignmentEnforced()
    Regime regime = TranslationRegime(PSTATE.EL);

    bit A;
    case regime of
        when Regime_EL3 A = SCTLR_EL3.A;
        when Regime_EL30 A = SCTLR.A;
        when Regime_EL2 A = if ELUsingAArch32(EL2) then HSCTLR.A else SCTLR_EL2.A;
        when Regime_EL20 A = SCTLR_EL2.A;
        when Regime_EL10 A = if ELUsingAArch32(EL1) then SCTLR.A else SCTLR_EL1.A;
        otherwise Unreachable();

    return A == '1';
```

## Library pseudocode for shared/functions/memory/Allocation

```
constant bits(2) MemHint_No = '00';    // No Read-Allocate, No Write-Allocate
constant bits(2) MemHint_WA = '01';    // No Read-Allocate, Write-Allocate
constant bits(2) MemHint_RA = '10';    // Read-Allocate, No Write-Allocate
constant bits(2) MemHint_RWA = '11';   // Read-Allocate, Write-Allocate
```

## Library pseudocode for shared/functions/memory/BigEndian

```
// BigEndian()
// =====

boolean BigEndian(AccessType acctype)
    boolean bigend;
    if HaveNV2Ext() && acctype == AccessType_NV2 then
        return SCTLR_EL2.EE == '1';

    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR_ELx[].EOE != '0');
    else
        bigend = (SCTLR_ELx[].EE != '0');
    return bigend;
```

## Library pseudocode for shared/functions/memory/BigEndianReverse

```
// BigEndianReverse()
// =====

bits(width) BigEndianReverse (bits(width) value)
    assert width IN {8, 16, 32, 64, 128};
    integer half = width DIV 2;
    if width == 8 then return value;
    return BigEndianReverse(value<half-1:0>) : BigEndianReverse(value<width-1:half>);
```

## Library pseudocode for shared/functions/memory/Cacheability

```
constant bits(2) MemAttr_NC = '00';    // Non-cacheable
constant bits(2) MemAttr_WT = '10';    // Write-through
constant bits(2) MemAttr_WB = '11';    // Write-back
```

## Library pseudocode for shared/functions/memory/CreateAccDescA32LSMD

```
// CreateAccDescA32LSMD()
// =====
// Access descriptor for A32 loads/store multiple general purpose registers

AccessDescriptor CreateAccDescA32LSMD(MemOp memop)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_GPR);

    accdesc.read          = memop == MemOp\_LOAD;
    accdesc.write         = memop == MemOp\_STORE;
    accdesc.pan           = TRUE;
    accdesc.a32lsmd       = TRUE;
    accdesc.transactional = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescASIMD

```
// CreateAccDescASIMD()
// =====
// Access descriptor for ASIMD&FP loads/stores

AccessDescriptor CreateAccDescASIMD(MemOp memop, boolean nontemporal, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_ASIMD);

    accdesc.nontemporal    = nontemporal;
    accdesc.read           = memop == MemOp\_LOAD;
    accdesc.write          = memop == MemOp\_STORE;
    accdesc.pan            = TRUE;
    accdesc.streamingsve   = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
        "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked = FALSE;
    else
        accdesc.tagchecked = tagchecked;
    accdesc.transactional  = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescASIMDAcqRel

```
// CreateAccDescASIMDAcqRel()
// =====
// Access descriptor for ASIMD&FP loads/stores with ordering semantics

AccessDescriptor CreateAccDescASIMDAcqRel(MemOp memop, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_ASIMD);

    accdesc.acqpc          = memop == MemOp_LOAD;
    accdesc.relsc          = memop == MemOp_STORE;
    accdesc.read           = memop == MemOp_LOAD;
    accdesc.write          = memop == MemOp_STORE;
    accdesc.pan            = TRUE;
    accdesc.streamingsve   = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
        "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked = FALSE;
    else
        accdesc.tagchecked = tagchecked;
    accdesc.transactional  = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescAT

```
// CreateAccDescAT()
// =====
// Access descriptor for address translation operations

AccessDescriptor CreateAccDescAT(SecurityState ss, bits(2) el, ATAccess ataccess)
    AccessDescriptor accdesc = NewAccDesc(AccessType_AT);

    accdesc.el            = el;
    accdesc.ss            = ss;
    case ataccess of
        when ATAccess_Read
            (accdesc.read, accdesc.write, accdesc.pan) = (TRUE, FALSE, FALSE);
        when ATAccess_ReadPAN
            (accdesc.read, accdesc.write, accdesc.pan) = (TRUE, FALSE, TRUE);
        when ATAccess_Write
            (accdesc.read, accdesc.write, accdesc.pan) = (FALSE, TRUE, FALSE);
        when ATAccess_WritePAN
            (accdesc.read, accdesc.write, accdesc.pan) = (FALSE, TRUE, TRUE);
        when ATAccess_Any
            (accdesc.read, accdesc.write, accdesc.pan) = (FALSE, FALSE, FALSE);

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescAcqRel

```
// CreateAccDescAcqRel()
// =====
// Access descriptor for general purpose register loads/stores with ordering semantics

AccessDescriptor CreateAccDescAcqRel(MemOp memop, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc         = memop == MemOp_LOAD;
    accdesc.relsc         = memop == MemOp_STORE;
    accdesc.read          = memop == MemOp_LOAD;
    accdesc.write          = memop == MemOp_STORE;
    accdesc.pan           = TRUE;
    accdesc.tagchecked     = tagchecked;
    accdesc.transactional  = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescAtomicOp

```
// CreateAccDescAtomicOp()
// =====
// Access descriptor for atomic read-modify-write memory accesses

AccessDescriptor CreateAccDescAtomicOp(MemAtomicOp modop, boolean acquire, boolean release,
                                       boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_GPR);

    accdesc.acqsc          = acquire;
    accdesc.relsc          = release;
    accdesc.atomicop       = TRUE;
    accdesc.modop          = modop;
    accdesc.read           = TRUE;
    accdesc.write          = TRUE;
    accdesc.pan            = TRUE;
    accdesc.tagchecked     = tagchecked;
    accdesc.transactional  = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescDC

```
// CreateAccDescDC()
// =====
// Access descriptor for data cache operations

AccessDescriptor CreateAccDescDC(CacheRecord cache)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_DC);

    accdesc.cacheop        = cache.cacheop;
    accdesc.cachetype      = cache.cachetype;
    accdesc.opscope        = cache.opscope;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescDCZero

```
// CreateAccDescDCZero()
// =====
// Access descriptor for data cache zero operations

AccessDescriptor CreateAccDescDCZero(boolean tagaccess, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_DCZero);

    accdesc.write          = TRUE;
    accdesc.pan            = TRUE;
    accdesc.tagchecked     = tagchecked;
    accdesc.tagaccess      = tagaccess;
    accdesc.transactional  = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescExLDST

```
// CreateAccDescExLDST()
// =====
// Access descriptor for general purpose register loads/stores with exclusive semantics

AccessDescriptor CreateAccDescExLDST(MemOp memop, boolean acqrel, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc          = acqrel && memop == MemOp_LOAD;
    accdesc.relsc          = acqrel && memop == MemOp_STORE;
    accdesc.exclusive      = TRUE;
    accdesc.read           = memop == MemOp_LOAD;
    accdesc.write          = memop == MemOp_STORE;
    accdesc.pan            = TRUE;
    accdesc.tagchecked     = tagchecked;
    accdesc.transactional  = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescGCS

```
// CreateAccDescGCS()
// =====
// Access descriptor for memory accesses to the Guarded Control Stack

AccessDescriptor CreateAccDescGCS(bits(2) el, MemOp memop)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GCS);

    accdesc.el             = el;
    accdesc.read           = memop == MemOp_LOAD;
    accdesc.write          = memop == MemOp_STORE;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescGCSSS1

```
// CreateAccDescGCSSS1()
// =====
// Access descriptor for memory accesses to the Guarded Control Stack that switch stacks

AccessDescriptor CreateAccDescGCSSS1(bits(2) el)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GCS);

    accdesc.el             = el;
    accdesc.atomicop       = TRUE;
    accdesc.modop          = MemAtomicOp_GCSSS1;
    accdesc.read           = TRUE;
    accdesc.write          = TRUE;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescGPR

```
// CreateAccDescGPR()
// =====
// Access descriptor for general purpose register loads/stores
// without exclusive or ordering semantics

AccessDescriptor CreateAccDescGPR(MemOp memop, boolean nontemporal, boolean privileged,
                                   boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_GPR);

    accdesc.el                = if !privileged then EL0 else PSTATE.EL;
    accdesc.nontemporal       = nontemporal;
    accdesc.read              = memop == MemOp\_LOAD;
    accdesc.write             = memop == MemOp\_STORE;
    accdesc.pan               = TRUE;
    accdesc.tagchecked        = tagchecked;
    accdesc.transactional     = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescGPTW

```
// CreateAccDescGPTW()
// =====
// Access descriptor for Granule Protection Table walks

AccessDescriptor CreateAccDescGPTW(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_GPTW);

    accdesc.el                = accdesc_in.el;
    accdesc.ss                = accdesc_in.ss;
    accdesc.read              = TRUE;
    accdesc.mpam              = accdesc_in.mpam;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescIC

```
// CreateAccDescIC()
// =====
// Access descriptor for instruction cache operations

AccessDescriptor CreateAccDescIC(CacheRecord cache)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_IC);

    accdesc.cacheop           = cache.cacheop;
    accdesc.cachetype         = cache.cachetype;
    accdesc.opscope           = cache.opscope;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescIFetch

```
// CreateAccDescIFetch()
// =====
// Access descriptor for instruction fetches

AccessDescriptor CreateAccDescIFetch()
    AccessDescriptor accdesc = NewAccDesc(AccessType\_IFETCH);

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescLDAcqPC

```
// CreateAccDescLDAcqPC()
// =====
// Access descriptor for general purpose register loads with local ordering semantics

AccessDescriptor CreateAccDescLDAcqPC(boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_GPR);

    accdesc.acqpc          = TRUE;
    accdesc.read           = TRUE;
    accdesc.pan            = TRUE;
    accdesc.tagchecked     = tagchecked;
    accdesc.transactional  = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescLDGSTG

```
// CreateAccDescLDGSTG()
// =====
// Access descriptor for tag memory loads/stores

AccessDescriptor CreateAccDescLDGSTG(MemOp memop)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_GPR);

    accdesc.read           = memop == MemOp\_LOAD;
    accdesc.write          = memop == MemOp\_STORE;
    accdesc.pan            = TRUE;
    accdesc.tagaccess      = TRUE;
    accdesc.transactional  = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescLOR

```
// CreateAccDescLOR()
// =====
// Access descriptor for general purpose register loads/stores with limited ordering semantics

AccessDescriptor CreateAccDescLOR(MemOp memop, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_GPR);

    accdesc.acqsc          = memop == MemOp\_LOAD;
    accdesc.relsc          = memop == MemOp\_STORE;
    accdesc.limitedordered = TRUE;
    accdesc.read           = memop == MemOp\_LOAD;
    accdesc.write          = memop == MemOp\_STORE;
    accdesc.pan            = TRUE;
    accdesc.tagchecked     = tagchecked;
    accdesc.transactional  = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```



## Library pseudocode for shared/functions/memory/CreateAccDescLS64

```
// CreateAccDescLS64()
// =====
// Access descriptor for accelerator-supporting memory accesses

AccessDescriptor CreateAccDescLS64(MemOp memop, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.read          = memop == MemOp_LOAD;
    accdesc.write         = memop == MemOp_STORE;
    accdesc.pan           = TRUE;
    accdesc.ls64          = TRUE;
    accdesc.tagchecked    = tagchecked;
    accdesc.transactional = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescMOPS

```
// CreateAccDescMOPS()
// =====
// Access descriptor for data memory copy and set instructions

AccessDescriptor CreateAccDescMOPS(MemOp memop, boolean privileged, boolean nontemporal)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.el            = if !privileged then EL0 else PSTATE.EL;
    accdesc.nontemporal   = nontemporal;
    accdesc.read          = memop == MemOp_LOAD;
    accdesc.write         = memop == MemOp_STORE;
    accdesc.pan           = TRUE;
    accdesc.mops          = TRUE;
    accdesc.tagchecked    = TRUE;
    accdesc.transactional = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescNV2

```
// CreateAccDescNV2()
// =====
// Access descriptor nested virtualization memory indirection loads/stores

AccessDescriptor CreateAccDescNV2(MemOp memop)
    AccessDescriptor accdesc = NewAccDesc(AccessType_NV2);

    accdesc.el            = EL2;
    accdesc.ss            = SecurityStateAtEL(EL2);
    accdesc.read          = memop == MemOp_LOAD;
    accdesc.write         = memop == MemOp_STORE;
    accdesc.transactional = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescRCW

```
// CreateAccDescRCW()
// =====
// Access descriptor for atomic read-check-write memory accesses

AccessDescriptor CreateAccDescRCW(MemAtomicOp modop, boolean soft, boolean acquire,
                                   boolean release, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc      = acquire;
    accdesc.relsc      = release;
    accdesc.rcw        = TRUE;
    accdesc.rcws       = soft;
    accdesc.atomicop    = TRUE;
    accdesc.modop       = modop;
    accdesc.read        = TRUE;
    accdesc.write       = TRUE;
    accdesc.pan         = TRUE;
    accdesc.tagchecked  = tagchecked;
    accdesc.transactional = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescS1TTW

```
// CreateAccDescS1TTW()
// =====
// Access descriptor for stage 1 translation table walks

AccessDescriptor CreateAccDescS1TTW(boolean toplevel, VARange varange, AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType_TTW);

    accdesc.el          = accdesc_in.el;
    accdesc.ss          = accdesc_in.ss;
    accdesc.read        = TRUE;
    accdesc.toplevel    = toplevel;
    accdesc.varange     = varange;
    accdesc.mpam        = accdesc_in.mpam;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescS2TTW

```
// CreateAccDescS2TTW()
// =====
// Access descriptor for stage 2 translation table walks

AccessDescriptor CreateAccDescS2TTW(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType_TTW);

    accdesc.el          = accdesc_in.el;
    accdesc.ss          = accdesc_in.ss;
    accdesc.read        = TRUE;
    accdesc.mpam        = accdesc_in.mpam;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescSME

```
// CreateAccDescSME()
// =====
// Access descriptor for SME loads/stores

AccessDescriptor CreateAccDescSME(MemOp memop, boolean nontemporal, boolean contiguous,
                                   boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_SME);

    accdesc.nontemporal      = nontemporal;
    accdesc.read             = memop == MemOp_LOAD;
    accdesc.write            = memop == MemOp_STORE;
    accdesc.pan              = TRUE;
    accdesc.contiguous       = contiguous;
    accdesc.streamingsve     = TRUE;
    if boolean IMPLEMENTATION_DEFINED "No tag checking of SME LDR & STR instructions" then
        accdesc.tagchecked   = FALSE;
    else
        accdesc.tagchecked   = tagchecked;
    accdesc.transactional     = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescSPE

```
// CreateAccDescSPE()
// =====
// Access descriptor for memory accesses by Statistical Profiling unit

AccessDescriptor CreateAccDescSPE(SecurityState owning_ss, bits(2) owning_el)
    AccessDescriptor accdesc = NewAccDesc(AccessType_SPE);

    accdesc.el              = owning_el;
    accdesc.ss              = owning_ss;
    accdesc.write           = TRUE;
    accdesc.mpm             = GenMPAMatEL(AccessType_SPE, owning_el);

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescSTGMOPS

```
// CreateAccDescSTGMOPS()
// =====
// Access descriptor for tag memory set instructions

AccessDescriptor CreateAccDescSTGMOPS(boolean privileged, boolean nontemporal)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.el              = if !privileged then ELO else PSTATE.EL;
    accdesc.nontemporal     = nontemporal;
    accdesc.write           = TRUE;
    accdesc.pan             = TRUE;
    accdesc.mops            = TRUE;
    accdesc.tagaccess       = TRUE;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescSVE

```
// CreateAccDescSVE()
// =====
// Access descriptor for general SVE loads/stores

AccessDescriptor CreateAccDescSVE(MemOp memop, boolean nontemporal, boolean contiguous,
                                   boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_SVE);

    accdesc.nontemporal      = nontemporal;
    accdesc.read             = memop == MemOp_LOAD;
    accdesc.write            = memop == MemOp_STORE;
    accdesc.pan              = TRUE;
    accdesc.contiguous       = contiguous;
    accdesc.streamingsve     = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
        "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked   = FALSE;
    else
        accdesc.tagchecked   = tagchecked;
    accdesc.transactional    = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescSVEFF

```
// CreateAccDescSVEFF()
// =====
// Access descriptor for first-fault SVE loads

AccessDescriptor CreateAccDescSVEFF(boolean contiguous, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_SVE);

    accdesc.read             = TRUE;
    accdesc.pan              = TRUE;
    accdesc.firstfault       = TRUE;
    accdesc.first            = TRUE;
    accdesc.contiguous       = contiguous;
    accdesc.streamingsve     = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
        "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked   = FALSE;
    else
        accdesc.tagchecked   = tagchecked;
    accdesc.transactional    = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescSVENF

```
// CreateAccDescSVENF()
// =====
// Access descriptor for non-fault SVE loads

AccessDescriptor CreateAccDescSVENF(boolean contiguous, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_SVE);

    accdesc.read          = TRUE;
    accdesc.pan           = TRUE;
    accdesc.nonfault      = TRUE;
    accdesc.contiguous    = contiguous;
    accdesc.streamingsve  = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
        "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked = FALSE;
    else
        accdesc.tagchecked = tagchecked;
    accdesc.transactional = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescTRBE

```
// CreateAccDescTRBE()
// =====
// Access descriptor for memory accesses by Trace Buffer Unit

AccessDescriptor CreateAccDescTRBE(SecurityState owning_ss, bits(2) owning_el)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_TRBE);

    accdesc.el           = owning_el;
    accdesc.ss           = owning_ss;
    accdesc.write        = TRUE;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescTTEUpdate

```
// CreateAccDescTTEUpdate()
// =====
// Access descriptor for translation table entry HW update

AccessDescriptor CreateAccDescTTEUpdate(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_TTW);

    accdesc.el           = accdesc_in.el;
    accdesc.ss           = accdesc_in.ss;
    accdesc.atomicop     = TRUE;
    accdesc.modop        = MemAtomicOp\_CAS;
    accdesc.read         = TRUE;
    accdesc.write        = TRUE;
    accdesc.mpam         = accdesc_in.mpam;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/DataMemoryBarrier

```
// DataMemoryBarrier()
// =====

DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);
```

## Library pseudocode for shared/functions/memory/DataSynchronizationBarrier

```
// DataSynchronizationBarrier()
// =====

DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types, boolean nXS);
```

## Library pseudocode for shared/functions/memory/DeviceType

```
// DeviceType
// =====
// Extended memory types for Device memory.

enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
```

## Library pseudocode for shared/functions/memory/EffectiveMTX

```
// EffectiveMTX()
// =====
// Returns the effective MTX in the AArch64 stage 1 translation regime for "el".

bit EffectiveMTX(bits(64) address, boolean is_instr, bits(2) el)
    bit mtx;
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    if !HaveMTE4Ext() || is_instr then
        mtx = '0';
    else
        case regime of
            when EL1
                mtx = if address<55> == '1' then TCR_EL1.MTX1 else TCR_EL1.MTX0;
            when EL2
                if HaveVirtHostExt() && ELIsInHost(el) then
                    mtx = if address<55> == '1' then TCR_EL2.MTX1 else TCR_EL2.MTX0;
                else
                    mtx = TCR_EL2.MTX;
            when EL3
                mtx = TCR_EL3.MTX;

    return mtx;
```

## Library pseudocode for shared/functions/memory/EffectiveTBI

```
// EffectiveTBI()
// =====
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

bit EffectiveTBI(bits(64) address, boolean IsInstr, bits(2) el)
    bit tbi;
    bit tbid;
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
            if HavePACEExt() then
                tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tbi = if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0;
                if HavePACEExt() then
                    tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
            else
                tbi = TCR_EL2.TBI;
                if HavePACEExt() then tbid = TCR_EL2.TBID;
        when EL3
            tbi = TCR_EL3.TBI;
            if HavePACEExt() then tbid = TCR_EL3.TBID;

    return (if tbi == '1' && (!HavePACEExt() || tbid == '0' || !IsInstr) then '1' else '0');
```

## Library pseudocode for shared/functions/memory/EffectiveTCMA

```
// EffectiveTCMA()
// =====
// Returns the effective TCMA of a virtual address in the stage 1 translation regime for "el".

bit EffectiveTCMA(bits(64) address, bits(2) el)
    bit tcma;
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tcma = if address<55> == '1' then TCR_EL1.TCMA1 else TCR_EL1.TCMA0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tcma = if address<55> == '1' then TCR_EL2.TCMA1 else TCR_EL2.TCMA0;
            else
                tcma = TCR_EL2.TCMA;
        when EL3
            tcma = TCR_EL3.TCMA;

    return tcma;
```

## Library pseudocode for shared/functions/memory/ErrorState

```
// ErrorState
// =====
// The allowed error states that can be returned by memory and used by the PE.

enumeration ErrorState {ErrorState_UC,           // Uncontainable
                        ErrorState_UEU,          // Unrecoverable state
                        ErrorState_UEO,          // Restartable state
                        ErrorState_UER,          // Recoverable state
                        ErrorState_CE,           // Corrected
                        ErrorState_Uncategorized,
                        ErrorState_IMPDEF};
```

## Library pseudocode for shared/functions/memory/Fault

```
// Fault
// =====
// Fault types.

enumeration Fault {Fault_None,
                  Fault_AccessFlag,
                  Fault_Alignment,
                  Fault_Background,
                  Fault_Domain,
                  Fault_Permission,
                  Fault_Translation,
                  Fault_AddressSize,
                  Fault_SyncExternal,
                  Fault_SyncExternalOnWalk,
                  Fault_SyncParity,
                  Fault_SyncParityOnWalk,
                  Fault_GPCFOnWalk,
                  Fault_GPCFOnOutput,
                  Fault_AsyncParity,
                  Fault_AsyncExternal,
                  Fault_TagCheck,
                  Fault_Debug,
                  Fault_TLBConflict,
                  Fault_BranchTarget,
                  Fault_HWUpdateAccessFlag,
                  Fault_Lockdown,
                  Fault_Exclusive,
                  Fault_ICacheMaint};
```



## Library pseudocode for shared/functions/memory/FaultRecord

```
// FaultRecord
// =====
// Fields that relate only to Faults.

type FaultRecord is (
    Fault          statuscode,          // Fault Status
    AccessDescriptor accessdesc,      // Details of the faulting access
    FullAddress    ipaddress,          // Intermediate physical address
    GPCFRecord     gpcf,               // Granule Protection Check Fault record
    FullAddress    paddress,          // Physical address
    boolean        gpcfs2walk,        // GPC for a stage 2 translation table walk
    boolean        s2fslwalk,        // Is on a Stage 1 translation table walk
    boolean        write,             // TRUE for a write, FALSE for a read
    boolean        sltagnotdata,      // TRUE for a fault due to tag not accessible at stage 1.
    boolean        tagaccess,         // TRUE for a fault due to NoTagAccess permission.
    integer        level,             // For translation, access flag and Permission faults
    bit            extflag,           // IMPLEMENTATION DEFINED syndrome for External aborts
    boolean        secondstage,       // Is a Stage 2 abort
    boolean        assuredonly,       // Stage 2 Permission fault due to AssuredOnly attribute
    boolean        toplevel,          // Stage 2 Permission fault due to TopLevel
    boolean        overlay,           // Fault due to overlay permissions
    boolean        dirtybit,          // Fault due to dirty state
    bits(4)        domain,            // Domain number, AArch32 only
    ErrorState     merrorstate,        // Incoming error state from memory
    boolean        maybe_false_match, // Watchpoint matches rounded range
    integer        watchpt_num,       // Matching watchpoint number
    bits(4)        debugmoe           // Debug method of entry, from AArch32 only
)
```

## Library pseudocode for shared/functions/memory/FullAddress

```
// FullAddress
// =====
// Physical or Intermediate Physical Address type.
// Although AArch32 only has access to 40 bits of physical or intermediate physical address space,
// the full address type has 56 bits to allow interprocessing with AArch64.
// The maximum physical or intermediate physical address size is IMPLEMENTATION DEFINED,
// but never exceeds 56 bits.

type FullAddress is (
    PASpace    paspace,
    bits(56)  address
)
```

## Library pseudocode for shared/functions/memory/GPCF

```
// GPCF
// ====
// Possible Granule Protection Check Fault reasons

enumeration GPCF {
    GPCF_None,          // No fault
    GPCF_AddressSize,   // GPT address size fault
    GPCF_Walk,          // GPT walk fault
    GPCF_EABT,          // Synchronous External abort on GPT fetch
    GPCF_Fail           // Granule protection fault
};
```

## Library pseudocode for shared/functions/memory/GPCFRecord

```
// GPCFRecord
// =====
// Full details of a Granule Protection Check Fault

type GPCFRecord is (
    GPCF    gpfc,
    integer level
)
```

## Library pseudocode for shared/functions/memory/Hint\_Prefetch

```
// Hint_Prefetch()
// =====
// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visible state should be on caches and TLBs associated with address, which must be
// accessible by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.

Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);
```

## Library pseudocode for shared/functions/memory/Hint\_RangePrefetch

```
// Hint_RangePrefetch()
// =====
// Signals the memory system that data memory accesses from a specified range
// of addresses are likely to occur in the near future. The memory system can
// respond by taking actions that are expected to speed up the memory accesses
// when they do occur, such as preloading the locations within the specified
// address ranges into one or more caches.

Hint_RangePrefetch(bits(64) address, integer length, integer stride,
    integer count, integer reuse, bits(6) operation);
```

## Library pseudocode for shared/functions/memory/IsDataAccess

```
// IsDataAccess()
// =====
// Return TRUE if access is to data memory.

boolean IsDataAccess(AccessType acctype)
    return !(acctype IN {AccessType\_IFETCH,
        AccessType\_TTW,
        AccessType\_DC,
        AccessType\_IC,
        AccessType\_AT});
```

## Library pseudocode for shared/functions/memory/IsSMEAccess

```
// IsSMEAccess()
// =====
// Return TRUE if access is of SME load/stores.

boolean IsSMEAccess(AccessDescriptor accdesc)
    return HaveSME() && accdesc.acctype == AccessType\_SME;
```

## Library pseudocode for shared/functions/memory/IsSVEAccess

```
// IsSVEAccess()
// =====
// Return TRUE if memory access is load/stores in an SVE mode.

boolean IsSVEAccess(AccessDescriptor accdesc)
    return HaveSVE() && accdesc.acctype == AccessType\_SVE;
```

## Library pseudocode for shared/functions/memory/MBReqDomain

```
// MBReqDomain
// =====
// Memory barrier domain.

enumeration MBReqDomain    {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
                           MBReqDomain_OuterShareable, MBReqDomain_FullSystem};
```

## Library pseudocode for shared/functions/memory/MBReqTypes

```
// MBReqTypes
// =====
// Memory barrier read/write.

enumeration MBReqTypes     {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```

## Library pseudocode for shared/functions/memory/MPAM

```
// MPAM Types
// =====

type PARTIDtype = bits(16);

type PMGtype = bits(8);

enumeration PARTIDspaceType {
    PIdSpace_Secure,
    PIdSpace_Root,
    PIdSpace_Realm,
    PIdSpace_NonSecure
};

type MPAMinfo is (
    PARTIDspaceType mpam_sp,
    PARTIDtype partid,
    PMGtype pmg
)
```

## Library pseudocode for shared/functions/memory/MemAtomicOp

```
// MemAtomicOp
// =====
// Atomic data processing instruction types.

enumeration MemAtomicOp {
    MemAtomicOp_GCSSS1,
    MemAtomicOp_ADD,
    MemAtomicOp_BIC,
    MemAtomicOp_EOR,
    MemAtomicOp_ORR,
    MemAtomicOp_SMAX,
    MemAtomicOp_SMIN,
    MemAtomicOp_UMAX,
    MemAtomicOp_UMIN,
    MemAtomicOp_SWP,
    MemAtomicOp_CAS
};

enumeration CacheOp {
    CacheOp_Clean,
    CacheOp_Invalidate,
    CacheOp_CleanInvalidate
};

enumeration CacheOpScope {
    CacheOpScope_SetWay,
    CacheOpScope_PoU,
    CacheOpScope_PoC,
    CacheOpScope_PoE,
    CacheOpScope_PoP,
    CacheOpScope_PoDP,
    CacheOpScope_PoPA,
    CacheOpScope_ALLU,
    CacheOpScope_ALLUIS
};

enumeration CacheType {
    CacheType_Data,
    CacheType_Tag,
    CacheType_Data_Tag,
    CacheType_Instruction
};

enumeration CachePASpace {
    CPAS_NonSecure,
    CPAS_Any, // Applicable only for DC *SW / IC IALLU* in Root state:
              // match entries from any PA Space
    CPAS_RealmNonSecure, // Applicable only for DC *SW / IC IALLU* in Realm state:
                        // match entries from Realm or Non-Secure PAS
    CPAS_Realm,
    CPAS_Root,
    CPAS_SecureNonSecure, // Applicable only for DC *SW / IC IALLU* in Secure state:
                        // match entries from Secure or Non-Secure PAS
    CPAS_Secure
};
```

## Library pseudocode for shared/functions/memory/MemAttrHints

```
// MemAttrHints
// =====
// Attributes and hints for Normal memory.

type MemAttrHints is (
    bits(2) attrs, // See MemAttr_*, Cacheability attributes
    bits(2) hints, // See MemHint_*, Allocation hints
    boolean transient
)
```

### Library pseudocode for shared/functions/memory/MemOp

```
// MemOp
// =====
// Memory access instruction types.

enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

### Library pseudocode for shared/functions/memory/MemType

```
// MemType
// =====
// Basic memory types.

enumeration MemType {MemType_Normal, MemType_Device};
```

### Library pseudocode for shared/functions/memory/Memory

```
// Memory Tag type
// =====

enumeration MemTagType {
    MemTag_Untagged,
    MemTag_AllocationTagged,
    MemTag_CanonicallyTagged
};
```

### Library pseudocode for shared/functions/memory/MemoryAttributes

```
// MemoryAttributes
// =====
// Memory attributes descriptor

type MemoryAttributes is (
    MemType      memtype,
    DeviceType   device,      // For Device memory types
    MemAttrHints inner,      // Inner hints and attributes
    MemAttrHints outer,      // Outer hints and attributes
    Shareability shareability, // Shareability attribute
    MemTagType   tags,        // MTE tag type for this memory.
    boolean      notagaccess, // Allocation Tag access permission
    bit          xs           // XS attribute
)
```

## Library pseudocode for shared/functions/memory/NewAccDesc

```
// NewAccDesc()
// =====
// Create a new AccessDescriptor with initialised fields

AccessDescriptor NewAccDesc(AccessType acctype)
    AccessDescriptor accdesc;

    accdesc.acctype          = acctype;
    accdesc.el               = PSTATE.EL;
    accdesc.ss               = SecurityStateAtEL(PSTATE.EL);
    accdesc.acqsc            = FALSE;
    accdesc.acqpc            = FALSE;
    accdesc.relsc            = FALSE;
    accdesc.limitedordered   = FALSE;
    accdesc.exclusive        = FALSE;
    accdesc.rcw              = FALSE;
    accdesc.rcws             = FALSE;
    accdesc.atomicop         = FALSE;
    accdesc.nontemporal      = FALSE;
    accdesc.read             = FALSE;
    accdesc.write            = FALSE;
    accdesc.pan              = FALSE;
    accdesc.nonfault         = FALSE;
    accdesc.firstfault       = FALSE;
    accdesc.first            = FALSE;
    accdesc.contiguous       = FALSE;
    accdesc.streamingsve     = FALSE;
    accdesc.ls64             = FALSE;
    accdesc.mops             = FALSE;
    accdesc.a32lsmd          = FALSE;
    accdesc.tagchecked        = FALSE;
    accdesc.tagaccess        = FALSE;
    accdesc.transactional    = FALSE;
    accdesc.mpam             = GenMPAMcurEL(acctype);
    accdesc.ispair           = FALSE;
    accdesc.highestaddressfirst = FALSE;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/PASpace

```
// PASpace
// =====
// Physical address spaces

enumeration PASpace {
    PAS_NonSecure,
    PAS_Secure,
    PAS_Root,
    PAS_Realm
};
```

## Library pseudocode for shared/functions/memory/Permissions

```
// Permissions
// =====
// Access Control bits in translation table descriptors

type Permissions is (
    bits(2) ap_table,    // Stage 1 hierarchical access permissions
    bit     xn_table,    // Stage 1 hierarchical execute-never for single EL regimes
    bit     pxn_table,   // Stage 1 hierarchical privileged execute-never
    bit     uxnn_table,  // Stage 1 hierarchical unprivileged execute-never
    bits(3) ap,          // Stage 1 access permissions
    bit     xn,          // Stage 1 execute-never for single EL regimes
    bit     uxnn,        // Stage 1 unprivileged execute-never
    bit     pxnn,        // Stage 1 privileged execute-never
    bits(4) ppi,         // Stage 1 privileged indirect permissions
    bits(4) upi,         // Stage 1 unprivileged indirect permissions
    bit     ndirty,      // Stage 1 dirty state for indirect permissions scheme
    bits(4) s2pi,        // Stage 2 indirect permissions
    bit     s2dirty,     // Stage 2 dirty state
    bits(4) po_index,    // Stage 1 overlay permissions index
    bits(4) s2po_index,  // Stage 2 overlay permissions index
    bits(2) s2ap,        // Stage 2 access permissions
    bit     s2tag_na,    // Stage 2 tag access
    bit     s2xnn,       // Stage 2 extended execute-never
    bit     s2xn         // Stage 2 execute-never
)
```

## Library pseudocode for shared/functions/memory/PhysMemRead

```
// PhysMemRead()
// =====
// Returns the value read from memory, and a status.
// Returned value is UNKNOWN if an External abort occurred while reading the
// memory.
// Otherwise the PhysMemRetStatus statuscode is Fault_None.

(PhysMemRetStatus, bits(8*size)) PhysMemRead(AddressDescriptor desc, integer size,
AccessDescriptor accdesc);
```

## Library pseudocode for shared/functions/memory/PhysMemRetStatus

```
// PhysMemRetStatus
// =====
// Fields that relate only to return values of PhysMem functions.

type PhysMemRetStatus is (
    Fault      statuscode,    // Fault Status
    bit        extflag,      // IMPLEMENTATION DEFINED syndrome for External aborts
    ErrorState merrorstate,  // Optional error state returned on a physical memory access
    bits(64)   store64bstatus // Status of 64B store
)
```

## Library pseudocode for shared/functions/memory/PhysMemWrite

```
// PhysMemWrite()
// =====
// Writes the value to memory, and returns the status of the write.
// If there is an External abort on the write, the PhysMemRetStatus indicates this.
// Otherwise the statuscode of PhysMemRetStatus is Fault_None.

PhysMemRetStatus PhysMemWrite(AddressDescriptor desc, integer size, AccessDescriptor accdesc,
bits(8*size) value);
```

## Library pseudocode for shared/functions/memory/PrefetchHint

```
// PrefetchHint
// =====
// Prefetch hint types.

enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
```

## Library pseudocode for shared/functions/memory/S1AccessControls

```
// S1AccessControls
// =====
// Effective access controls defined by stage 1 translation

type S1AccessControls is (
    bit r,                // Stage 1 base read permission
    bit w,                // Stage 1 base write permission
    bit x,                // Stage 1 base execute permission
    bit gcs,              // Stage 1 GCS permission
    boolean overlay,      // Stage 1 overlay feature enabled
    bit or,               // Stage 1 overlay read permission
    bit ow,               // Stage 1 overlay write permission
    bit ox,               // Stage 1 overlay execute permission
    bit wxn               // Stage 1 write permission implies execute-never
)
```

## Library pseudocode for shared/functions/memory/S2AccessControls

```
// S2AccessControls
// =====
// Effective access controls defined by stage 2 translation

type S2AccessControls is (
    bit r,                // Stage 2 read permission.
    bit w,                // Stage 2 write permission.
    bit x,                // Stage 2 execute permission.
    bit r_rcw,            // Stage 2 Read perms for RCW instruction.
    bit w_rcw,            // Stage 2 Write perms for RCW instruction.
    bit r_mmu,            // Stage 2 Read perms for TTW data.
    bit w_mmu,            // Stage 2 Write perms for TTW data.
    bit toplevel0,        // IPA as top level table for TTBR0_EL1.
    bit toplevel1,        // IPA as top level table for TTBR1_EL1.
    boolean overlay,      // Overlay enable
    bit or,               // Stage 2 overlay read permission.
    bit ow,               // Stage 2 overlay write permission.
    bit ox,               // Stage 2 overlay execute permission.
    bit or_rcw,           // Stage 2 overlay Read perms for RCW instruction.
    bit ow_rcw,           // Stage 2 overlay Write perms for RCW instruction.
    bit or_mmu,           // Stage 2 overlay Read perms for TTW data.
    bit ow_mmu,           // Stage 2 overlay Write perms for TTW data.
)
```

## Library pseudocode for shared/functions/memory/Shareability

```
// Shareability
// =====

enumeration Shareability {
    Shareability_NSH,
    Shareability_ISH,
    Shareability_OSH
};
```



### Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToPA

```
// SpeculativeStoreBypassBarrierToPA()
// =====

SpeculativeStoreBypassBarrierToPA();
```

### Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToVA

```
// SpeculativeStoreBypassBarrierToVA()
// =====

SpeculativeStoreBypassBarrierToVA();
```

### Library pseudocode for shared/functions/memory/Tag

```
constant integer LOG2_TAG_GRANULE = 4;

constant integer TAG_GRANULE = 1 << LOG2\_TAG\_GRANULE;
```

### Library pseudocode for shared/functions/memory/VARange

```
// VARange
// =====
// Virtual address ranges

enumeration VARange {
    VARange_LOWER,
    VARange_UPPER
};
```

### Library pseudocode for shared/functions/mpam/AltPARTIDspace

```
// AltPARTIDspace()
// =====
// From the Security state, EL and ALTSP configuration, determine
// whether to primary space or the alt space is selected and which
// PARTID space is the alternative space. Return that alternative
// PARTID space if selected or the primary space if not.

PARTIDspaceType AltPARTIDspace(bits(2) el, SecurityState security,
                                PARTIDspaceType primaryPidSpace)

case security of
when SS\_NonSecure
    assert el != EL3;
    return primaryPidSpace;
when SS\_Secure
    assert el != EL3;
    if primaryPidSpace == PidSpace\_NonSecure then
        return primaryPidSpace;
    return AltPidSecure(el, primaryPidSpace);
when SS\_Root
    assert el == EL3;
    if MPAM3_EL3.ALTSP_EL3 == '1' then
        if MPAM3_EL3.RT_ALTSP_NS == '1' then
            return PidSpace\_NonSecure;
        else
            return PidSpace\_Secure;
    else
        return primaryPidSpace;
when SS\_Realm
    assert el != EL3;
    return AltPidRealm(el, primaryPidSpace);
otherwise
    Unreachable();
```

## Library pseudocode for shared/functions/mpam/AltPidRealm

```
// AltPidRealm()
// =====
// Compute PARTID space as either the primary PARTID space or
// alternative PARTID space in the Realm Security state.
// Helper for AltPARTIDspace.

PARTIDspaceType AltPidRealm(bits(2) el, PARTIDspaceType primaryPidSpace)
PARTIDspaceType PIdSpace = primaryPidSpace;
case el of
  when EL0
    if ELIsInHost(EL0) then
      if !UsePrimarySpaceEL2() then
        PIdSpace = PIdSpace\_NonSecure;
      elseif !UsePrimarySpaceEL10() then
        PIdSpace = PIdSpace\_NonSecure;
  when EL1
    if !UsePrimarySpaceEL10() then
      PIdSpace = PIdSpace\_NonSecure;
  when EL2
    if !UsePrimarySpaceEL2() then
      PIdSpace = PIdSpace\_NonSecure;
  otherwise
    Unreachable();
return PIdSpace;
```

## Library pseudocode for shared/functions/mpam/AltPidSecure

```
// AltPidSecure()
// =====
// Compute PARTID space as either the primary PARTID space or
// alternative PARTID space in the Secure Security state.
// Helper for AltPARTIDspace.

PARTIDspaceType AltPidSecure(bits(2) el, PARTIDspaceType primaryPidSpace)
PARTIDspaceType PIdSpace = primaryPidSpace;
boolean el2en = EL2Enabled();
case el of
  when EL0
    if el2en then
      if ELIsInHost(EL0) then
        if !UsePrimarySpaceEL2() then
          PIdSpace = PIdSpace\_NonSecure;
        elseif !UsePrimarySpaceEL10() then
          PIdSpace = PIdSpace\_NonSecure;
      elseif MPAM3_EL3.ALTSP_HEN == '0' && MPAM3_EL3.ALTSP_HFC == '1' then
        PIdSpace = PIdSpace\_NonSecure;
  when EL1
    if el2en then
      if !UsePrimarySpaceEL10() then
        PIdSpace = PIdSpace\_NonSecure;
      elseif MPAM3_EL3.ALTSP_HEN == '0' && MPAM3_EL3.ALTSP_HFC == '1' then
        PIdSpace = PIdSpace\_NonSecure;
  when EL2
    if !UsePrimarySpaceEL2() then
      PIdSpace = PIdSpace\_NonSecure;
  otherwise
    Unreachable();
return PIdSpace;
```

### Library pseudocode for shared/functions/mpam/DefaultMPAMInfo

```
// DefaultMPAMInfo()
// =====
// Returns default MPAM info.  The partidspace argument sets
// the PARTID space of the default MPAM information returned.

MPAMInfo DefaultMPAMInfo(PARTIDspaceType partidspace)
    MPAMInfo DefaultInfo;
    DefaultInfo.mpam_sp = partidspace;
    DefaultInfo.partid  = DefaultPARTID;
    DefaultInfo.pmg     = DefaultPMG;
    return DefaultInfo;
```

### Library pseudocode for shared/functions/mpam/DefaultPARTID

```
constant PARTIDtype DefaultPARTID = 0<15:0>;
```

### Library pseudocode for shared/functions/mpam/DefaultPMG

```
constant PMGtype      DefaultPMG = 0<7:0>;
```

## Library pseudocode for shared/functions/mpam/GenMPAMatEL

```
// GenMPAMatEL()
// =====
// Returns MPAMinfo for the specified EL.
// May be called if MPAM is not implemented (but in a version that supports
// MPAM), MPAM is disabled, or in AArch32. In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation. If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo GenMPAMatEL(AccessType acctype, bits(2) el)
    bits(2) mpamEL;
    boolean validEL = FALSE;
    SecurityState security = SecurityStateAtEL(el);
    boolean InD = FALSE;
    boolean InSM = FALSE;
    PARTIDspaceType pspace = PARTIDspaceFromSS(security);
    if pspace == PIdSpace\_NonSecure && !MPAMisEnabled() then
        return DefaultMPAMinfo(pspace);
    if UsingAArch32() then
        (validEL, mpamEL) = ELFromM32(PSTATE.M);
    else
        mpamEL = if acctype == AccessType\_NV2 then EL2 else el;
        validEL = TRUE;
    case acctype of
        when AccessType\_IFETCH, AccessType\_IC
            InD = TRUE;
        when AccessType\_SME
            InSM = (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
                    boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1 label precedence");
        when AccessType\_ASIMD
            InSM = (HaveSME() && PSTATE.SM == '1' &&
                    (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
                     boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1 label precedence"));
        when AccessType\_SVE
            InSM = (HaveSME() && PSTATE.SM == '1' &&
                    (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
                     boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1 label precedence"));
        otherwise
            // Other access types are DATA accesses
            InD = FALSE;
    if !validEL then
        return DefaultMPAMinfo(pspace);
    elsif HaveRME() && MPAMIDR_EL1.HAS_ALTSP == '1' then
        // Substitute alternative PARTID space if selected
        pspace = AltPARTIDspace(mpamEL, security, pspace);
    if HaveMPAMv0p1Ext() && MPAMIDR_EL1.HAS_FORCE_NS == '1' then
        if MPAM3_EL3.FORCE_NS == '1' && security == SS\_Secure then
            pspace = PIdSpace\_NonSecure;
    if (HaveMPAMv0p1Ext() || HaveMPAMv1p1Ext()) && MPAMIDR_EL1.HAS_SDEFLT == '1' then
        if MPAM3_EL3.SDEFLT == '1' && security == SS\_Secure then
            return DefaultMPAMinfo(pspace);
    if !MPAMisEnabled() then
        return DefaultMPAMinfo(pspace);
    else
        return genMPAM(mpamEL, InD, InSM, pspace);
```

## Library pseudocode for shared/functions/mpam/GenMPAMcurEL

```
// GenMPAMcurEL()
// =====
// Returns MPAMinfo for the current EL and security state.
// May be called if MPAM is not implemented (but in a version that supports
// MPAM), MPAM is disabled, or in AArch32. In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation. If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo GenMPAMcurEL(AccessType acctype)
    return GenMPAMatEL(acctype, PSTATE.EL);
```

## Library pseudocode for shared/functions/mpam/MAP\_vPARTID

```
// MAP_vPARTID()
// =====
// Performs conversion of virtual PARTID into physical PARTID
// Contains all of the error checking and implementation
// choices for the conversion.

(PARTIDtype, boolean) MAP_vPARTID(PARTIDtype vpartid)
    // should not ever be called if EL2 is not implemented
    // or is implemented but not enabled in the current
    // security state.
    PARTIDtype ret;
    boolean err;
    integer virt = UInt(vpartid);
    integer vpmrmax = UInt(MPAMIDR_EL1.VPMR_MAX);

    // vpartid_max is largest vpartid supported
    integer vpartid_max = (vpmrmax << 2) + 3;

    // One of many ways to reduce vpartid to value less than vpartid_max.
    if UInt(vpartid) > vpartid_max then
        virt = virt MOD (vpartid_max+1);

    // Check for valid mapping entry.
    if MPAMVPMV_EL2<virt> == '1' then
        // vpartid has a valid mapping so access the map.
        ret = mapvpmw(virt);
        err = FALSE;

    // Is the default virtual PARTID valid?
    elsif MPAMVPMV_EL2<0> == '1' then
        // Yes, so use default mapping for vpartid == 0.
        ret = MPAMVPM0_EL2<0 +: 16>;
        err = FALSE;

    // Neither is valid so use default physical PARTID.
    else
        ret = DefaultPARTID;
        err = TRUE;

    // Check that the physical PARTID is in-range.
    // This physical PARTID came from a virtual mapping entry.
    integer partid_max = UInt(MPAMIDR_EL1.PARTID_MAX);
    if UInt(ret) > partid_max then
        // Out of range, so return default physical PARTID
        ret = DefaultPARTID;
        err = TRUE;
    return (ret, err);
```

## Library pseudocode for shared/functions/mpam/MPAMisEnabled

```
// MPAMisEnabled()
// =====
// Returns TRUE if MPAMisEnabled.

boolean MPAMisEnabled()
    el = HighestEL\(\);
    case el of
        when EL3 return MPAM3_EL3.MPAMEN == '1';
        when EL2 return MPAM2_EL2.MPAMEN == '1';
        when EL1 return MPAM1_EL1.MPAMEN == '1';
```

## Library pseudocode for shared/functions/mpam/MPAMisVirtual

```
// MPAMisVirtual()
// =====
// Returns TRUE if MPAM is configured to be virtual at EL.

boolean MPAMisVirtual(bits(2) el)
    return (MPAMIDR_EL1.HAS_HCR == '1' && EL2Enabled\(\) &&
        ((el == EL0 && MPAMHCR_EL2.EL0_VPMEN == '1' &&
            (HCR_EL2.E2H == '0' || HCR_EL2.TGE == '0')) ||
            (el == EL1 && MPAMHCR_EL2.EL1_VPMEN == '1')));
```

## Library pseudocode for shared/functions/mpam/PARTIDspaceFromSS

```
// PARTIDspaceFromSS()
// =====
// Returns the primary PARTID space from the Security State.

PARTIDspaceType PARTIDspaceFromSS(SecurityState security)
    case security of
        when SS\_NonSecure
            return PIdSpace\_NonSecure;
        when SS\_Root
            return PIdSpace\_Root;
        when SS\_Realm
            return PIdSpace\_Realm;
        when SS\_Secure
            return PIdSpace\_Secure;
        otherwise
            Unreachable\(\);
```

## Library pseudocode for shared/functions/mpam/UsePrimarySpaceEL10

```
// UsePrimarySpaceEL10()
// =====
// Checks whether Primary space is configured in the
// MPAM3_EL3 and MPAM2_EL2 ALTSP control bits that affect
// MPAM ALTSP use at EL1 and EL0.

boolean UsePrimarySpaceEL10()
    if MPAM3_EL3.ALTSP_HEN == '0' then
        return MPAM3_EL3.ALTSP_HFC == '0';
    return !MPAMisEnabled\(\) || !EL2Enabled\(\) || MPAM2_EL2.ALTSP_HFC == '0';
```

## Library pseudocode for shared/functions/mpam/UsePrimarySpaceEL2

```
// UsePrimarySpaceEL2()
// =====
// Checks whether Primary space is configured in the
// MPAM3_EL3 and MPAM2_EL2 ALTSP control bits that affect
// MPAM ALTSP use at EL2.

boolean UsePrimarySpaceEL2()
    if MPAM3_EL3.ALTSP_HEN == '0' then
        return MPAM3_EL3.ALTSP_HFC == '0';
    return !MPAMisEnabled() || MPAM2_EL2.ALTSP_EL2 == '0';
```

## Library pseudocode for shared/functions/mpam/genMPAM

```
// genMPAM()
// =====
// Returns MPAMinfo for exception level el.
// If InD is TRUE returns MPAM information using PARTID_I and PMG_I fields
// of MPAMel_ELx register and otherwise using PARTID_D and PMG_D fields.
// If InSM is TRUE returns MPAM information using PARTID_D and PMG_D fields
// of MPAMSM_EL1 register.
// Produces a PARTID in PARTID space pspace.

MPAMinfo genMPAM(bits(2) el, boolean InD, boolean InSM, PARTIDspaceType pspace)
    MPAMinfo returninfo;
    PARTIDtype partidel;
    boolean perr;
    // gstplk is guest OS application locked by the EL2 hypervisor to
    // only use EL1 the virtual machine's PARTIDs.
    boolean gstplk = (el == EL0 && EL2Enabled() &&
        MPAMHCR_EL2.GSTAPP_PLK == '1' &&
        HCR_EL2.TGE == '0');
    bits(2) eff_el = if gstplk then EL1 else el;
    (partidel, perr) = genPARTID(eff_el, InD, InSM);
    PMGtype groupel = genPMG(eff_el, InD, InSM, perr);
    returninfo.mpam_sp = pspace;
    returninfo.partid = partidel;
    returninfo.pmg = groupel;
    return returninfo;
```

## Library pseudocode for shared/functions/mpam/genPARTID

```
// genPARTID()
// =====
// Returns physical PARTID and error boolean for exception level el.
// If InD is TRUE then PARTID is from MPAMel_ELx.PARTID_I and
// otherwise from MPAMel_ELx.PARTID_D.
// If InSM is TRUE then PARTID is from MPAMSM_EL1.PARTID_D.

(PARTIDtype, boolean) genPARTID(bits(2) el, boolean InD, boolean InSM)
    PARTIDtype partidel = getMPAM_PARTID(el, InD, InSM);
    PARTIDtype partid_max = MPAMIDR_EL1.PARTID_MAX;
    if UInt(partidel) > UInt(partid_max) then
        return (DefaultPARTID, TRUE);
    if MPAMisVirtual(el) then
        return MAP_vPARTID(partidel);
    else
        return (partidel, FALSE);
```

## Library pseudocode for shared/functions/mpam/genPMG

```
// genPMG()
// =====
// Returns PMG for exception level el and I- or D-side (InD).
// If PARTID generation (genPARTID) encountered an error, genPMG() should be
// called with partid_err as TRUE.

PMGtype genPMG(bits(2) el, boolean InD, boolean InSM, boolean partid_err)
    integer pmg_max = UInt(MPAMIDR_EL1.PMG_MAX);
    // It is CONSTRAINED UNPREDICTABLE whether partid_err forces PMG to
    // use the default or if it uses the PMG from getMPAM_PMG.
    if partid_err then
        return DefaultPMG;
    PMGtype groupel = getMPAM_PMG(el, InD, InSM);
    if UInt(groupel) <= pmg_max then
        return groupel;
    return DefaultPMG;
```

## Library pseudocode for shared/functions/mpam/getMPAM\_PARTID

```
// getMPAM_PARTID()
// =====
// Returns a PARTID from one of the MPAMn_ELx or MPAMSM_EL1 registers.
// If InSM is TRUE, the MPAMSM_EL1 register is used. Otherwise,
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PARTID_I field of that
// register. Otherwise, selects the PARTID_D field.

PARTIDtype getMPAM_PARTID(bits(2) MPAMn, boolean InD, boolean InSM)
    PARTIDtype partid;
    boolean el2avail = EL2Enabled();

    if InSM then
        partid = MPAMSM_EL1.PARTID_D;
        return partid;

    if InD then
        case MPAMn of
            when '11' partid = MPAM3_EL3.PARTID_I;
            when '10' partid = if el2avail then MPAM2_EL2.PARTID_I else Zeros(16);
            when '01' partid = MPAM1_EL1.PARTID_I;
            when '00' partid = MPAM0_EL1.PARTID_I;
            otherwise partid = PARTIDtype UNKNOWN;
    else
        case MPAMn of
            when '11' partid = MPAM3_EL3.PARTID_D;
            when '10' partid = if el2avail then MPAM2_EL2.PARTID_D else Zeros(16);
            when '01' partid = MPAM1_EL1.PARTID_D;
            when '00' partid = MPAM0_EL1.PARTID_D;
            otherwise partid = PARTIDtype UNKNOWN;
    return partid;
```



## Library pseudocode for shared/functions/mpam/getMPAM\_PMG

```
// getMPAM_PMG()
// =====
// Returns a PMG from one of the MPAMn_ELx or MPAMSM_EL1 registers.
// If InSM is TRUE, the MPAMSM_EL1 register is used. Otherwise,
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PMG_I field of that
// register. Otherwise, selects the PMG_D field.

PMGtype getMPAM_PMG(bits(2) MPAMn, boolean InD, boolean InSM)
    PMGtype pmg;
    boolean el2avail = EL2Enabled();

    if InSM then
        pmg = MPAMSM_EL1.PMG_D;
        return pmg;

    if InD then
        case MPAMn of
            when '11' pmg = MPAM3_EL3.PMG_I;
            when '10' pmg = if el2avail then MPAM2_EL2.PMG_I else Zeros(8);
            when '01' pmg = MPAM1_EL1.PMG_I;
            when '00' pmg = MPAM0_EL1.PMG_I;
            otherwise pmg = PMGtype UNKNOWN;
        else
            case MPAMn of
                when '11' pmg = MPAM3_EL3.PMG_D;
                when '10' pmg = if el2avail then MPAM2_EL2.PMG_D else Zeros(8);
                when '01' pmg = MPAM1_EL1.PMG_D;
                when '00' pmg = MPAM0_EL1.PMG_D;
                otherwise pmg = PMGtype UNKNOWN;
            return pmg;
```

## Library pseudocode for shared/functions/mpam/mapvpmw

```
// mapvpmw()
// =====
// Map a virtual PARTID into a physical PARTID using
// the MPAMVPMn_EL2 registers.
// vpartid is now assumed in-range and valid (checked by caller)
// returns physical PARTID from mapping entry.

PARTIDtype mapvpmw(integer vpartid)
    bits(64) vpmw;
    integer wd = vpartid DIV 4;
    case wd of
        when 0 vpmw = MPAMVPM0_EL2;
        when 1 vpmw = MPAMVPM1_EL2;
        when 2 vpmw = MPAMVPM2_EL2;
        when 3 vpmw = MPAMVPM3_EL2;
        when 4 vpmw = MPAMVPM4_EL2;
        when 5 vpmw = MPAMVPM5_EL2;
        when 6 vpmw = MPAMVPM6_EL2;
        when 7 vpmw = MPAMVPM7_EL2;
        otherwise vpmw = Zeros(64);
    // vpme_lsb selects LSB of field within register
    integer vpme_lsb = (vpartid MOD 4) * 16;
    return vpmw < vpme_lsb +: 16>;
```

## Library pseudocode for shared/functions/predictionrestrict/ASID

```
// ASID[]
// =====
// Effective ASID.

bits(16) ASID[]
    if EL2Enabled\(\) && !ELUsingAArch32\(EL2\) && HCR_EL2.<E2H, TGE> == '11' then
        if TCR_EL2.A1 == '1' then
            return TTBR1_EL2.ASID;
        else
            return TTBR0_EL2.ASID;

    elseif !ELUsingAArch32\(EL1\) then
        if TCR_EL1.A1 == '1' then
            return TTBR1_EL1.ASID;
        else
            return TTBR0_EL1.ASID;

    else
        if TTBCR.EAE == '0' then
            return ZeroExtend(CONTEXTIDR.ASID, 16);
        else
            if TTBCR.A1 == '1' then
                return ZeroExtend(TTBR1.ASID, 16);
            else
                return ZeroExtend(TTBR0.ASID, 16);
```

## Library pseudocode for shared/functions/predictionrestrict/ExecutionCntxt

```
// ExecutionCntxt
// =====
// Context information for prediction restriction operation.

type ExecutionCntxt is (
    boolean        is_vmid_valid, // is vmid valid for current context
    boolean        all_vmid,      // should the operation be applied for all vmids
    bits(16)       vmid,          // if all_vmid = FALSE, vmid to which operation is applied
    boolean        is_asid_valid, // is asid valid for current context
    boolean        all_asid,      // should the operation be applied for all asids
    bits(16)       asid,          // if all_asid = FALSE, ASID to which operation is applied
    bits(2)        target_el,     // target EL at which operation is performed
    SecurityState  security,
    RestrictType   restriction    // type of restriction operation
)
```

## Library pseudocode for shared/functions/predictionrestrict/RESTRICT\_PREDICTIONS

```
// RESTRICT_PREDICTIONS()
// =====
// Clear all speculated values.

RESTRICT_PREDICTIONS(ExecutionCntxt c)
    IMPLEMENTATION_DEFINED;
```

## Library pseudocode for shared/functions/predictionrestrict/RestrictType

```
// RestrictType
// =====
// Type of restriction on speculation.

enumeration RestrictType {
    RestrictType_DataValue,
    RestrictType_ControlFlow,
    RestrictType_CachePrefetch,
    RestrictType_Other          // Any other trained speculation mechanisms than those above
};
```

## Library pseudocode for shared/functions/predictionrestrict/TargetSecurityState

```
// TargetSecurityState()
// =====
// Decode the target security state for the prediction context.

SecurityState TargetSecurityState(bit NS, bit NSE)
    curr_ss = SecurityStateAtEL(PSTATE.EL);
    if curr_ss == SS\_NonSecure then
        return SS\_NonSecure;
    elsif curr_ss == SS\_Secure then
        case NS of
            when '0' return SS\_Secure;
            when '1' return SS\_NonSecure;
    elsif HaveRME() then
        if curr_ss == SS\_Root then
            case NSE:NS of
                when '00' return SS\_Secure;
                when '01' return SS\_NonSecure;
                when '11' return SS\_Realm;
                when '10' return SS\_Root;
        elsif curr_ss == SS\_Realm then
            return SS\_Realm;
```

## Library pseudocode for shared/functions/registers/BranchTo

```
// BranchTo()
// =====
// Set program counter to a new address, with a branch type.
// Parameter branch_conditional indicates whether the executed branch has a conditional encoding.
// In AArch64 state the address might include a tag in the top eight bits.

BranchTo(bits(N) target, BranchType branch_type, boolean branch_conditional)
    Hint\_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target, 64);
    else
        assert N == 64 && !UsingAArch32();
        bits(64) target_vaddress = AArch64.BranchAddr(target<63:0>, PSTATE.EL);
        if (HaveBRBExt() &&
            branch_type IN {BranchType\_DIR, BranchType\_INDIR,
                           BranchType\_DIRCALL, BranchType\_INDCALL,
                           BranchType\_RET}) then
            BRBEBranch(branch_type, branch_conditional, target_vaddress);
            boolean branch_taken = TRUE;

            if HaveStatisticalProfiling() then
                SPEBranch(target, branch_type, branch_conditional, branch_taken);

        _PC = target_vaddress;
    return;
```

## Library pseudocode for shared/functions/registers/BranchToAddr

```
// BranchToAddr()
// =====
// Set program counter to a new address, with a branch type.
// In AArch64 state the address does not include a tag in the top eight bits.

BranchToAddr(bits(N) target, BranchType branch_type)
    Hint\_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target, 64);
    else
        assert N == 64 && !UsingAArch32();
        _PC = target<63:0>;
    return;
```

## Library pseudocode for shared/functions/registers/BranchType

```
// BranchType
// =====
// Information associated with a change in control flow.

enumeration BranchType {
    BranchType_DIRCALL,    // Direct Branch with link
    BranchType_INDICAL,    // Indirect Branch with link
    BranchType_ERET,       // Exception return (indirect)
    BranchType_DBGEXIT,    // Exit from Debug state
    BranchType_RET,        // Indirect branch with function return hint
    BranchType_DIR,        // Direct branch
    BranchType_INDIR,      // Indirect branch
    BranchType_EXCEPTION,  // Exception entry
    BranchType_TMFAIL,     // Transaction failure
    BranchType_RESET,      // Reset
    BranchType_UNKNOWN};   // Other
```

## Library pseudocode for shared/functions/registers/Hint\_Branch

```
// Hint_Branch()
// =====
// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction.

Hint_Branch(BranchType hint);
```

## Library pseudocode for shared/functions/registers/NextInstrAddr

```
// NextInstrAddr()
// =====
// Return address of the sequentially next instruction.

bits(N) NextInstrAddr(integer N);
```

## Library pseudocode for shared/functions/registers/ResetExternalDebugRegisters

```
// ResetExternalDebugRegisters()
// =====
// Reset the External Debug registers in the Core power domain.

ResetExternalDebugRegisters(boolean cold_reset);
```

## Library pseudocode for shared/functions/registers/ThisInstrAddr

```
// ThisInstrAddr()
// =====
// Return address of the current instruction.

bits(N) ThisInstrAddr(integer N)
    assert N == 64 || (N == 32 && UsingAArch32\(\));
    return _PC<N-1:0>;
```

## Library pseudocode for shared/functions/registers/\_PC

```
bits(64) _PC;
```

## Library pseudocode for shared/functions/registers/\_R

```
// _R[] - the general-purpose register file
// =====

array bits(64) _R[0..30];
```

## Library pseudocode for shared/functions/sysregisters/SPSR\_ELx

```
// SPSR_ELx[] - non-assignment form
// =====

bits(64) SPSR_ELx[]
    bits(64) result;
    case PSTATE.EL of
        when EL1      result = SPSR_EL1<63:0>;
        when EL2      result = SPSR_EL2<63:0>;
        when EL3      result = SPSR_EL3<63:0>;
        otherwise    Unreachable();
    return result;

// SPSR_ELx[] - assignment form
// =====

SPSR_ELx[] = bits(64) value
    case PSTATE.EL of
        when EL1      SPSR_EL1<63:0> = value<63:0>;
        when EL2      SPSR_EL2<63:0> = value<63:0>;
        when EL3      SPSR_EL3<63:0> = value<63:0>;
        otherwise    Unreachable();
    return;
```

## Library pseudocode for shared/functions/sysregisters/SPSR\_curr

```
// SPSR_curr[] - non-assignment form
// =====

bits(32) SPSR_curr[]
  bits(32) result;
  case PSTATE.M of
    when M32\_FIQ      result = SPSR_fiq<31:0>;
    when M32\_IRQ      result = SPSR_irq<31:0>;
    when M32\_Svc      result = SPSR_svc<31:0>;
    when M32\_Monitor  result = SPSR_mon<31:0>;
    when M32\_Abort    result = SPSR_abt<31:0>;
    when M32\_Hyp      result = SPSR_hyp<31:0>;
    when M32\_Undef    result = SPSR_und<31:0>;
    otherwise         Unreachable();
  return result;

// SPSR_curr[] - assignment form
// =====

SPSR_curr[] = bits(32) value
  case PSTATE.M of
    when M32\_FIQ      SPSR_fiq<31:0> = value<31:0>;
    when M32\_IRQ      SPSR_irq<31:0> = value<31:0>;
    when M32\_Svc      SPSR_svc<31:0> = value<31:0>;
    when M32\_Monitor  SPSR_mon<31:0> = value<31:0>;
    when M32\_Abort    SPSR_abt<31:0> = value<31:0>;
    when M32\_Hyp      SPSR_hyp<31:0> = value<31:0>;
    when M32\_Undef    SPSR_und<31:0> = value<31:0>;
    otherwise         Unreachable();
  return;
```

## Library pseudocode for shared/functions/system/AArch64.ChkFeat

```
// AArch64.ChkFeat()
// =====
// Indicates the status of some features

bits(64) AArch64.ChkFeat(bits(64) feat_select)
  bits(64) feat_en = Zeros(64);
  feat_en[0] = if HaveGCS() && GCSEnabled(PSTATE.EL) then '1' else '0';
  return feat_select AND NOT(feat_en);
```

## Library pseudocode for shared/functions/system/AddressNotInNaturallyAlignedBlock

```
// AddressNotInNaturallyAlignedBlock()
// =====
// The 'address' is not in a naturally aligned block if it doesn't meet all the below conditions:
// * is a power-of-two size.
// * Is no larger than the DC ZVA block size if ESR_ELx.FnP is being set to 0b0, or EDHSR is not
//   implemented or EDHSR.FnP is being set to 0b0 (as appropriate).
// * Is no larger than the smallest implemented translation granule if ESR_ELx.FnP, or EDHSR.FnP
//   (as appropriate) is being set to 0b1.
// * Contains a watchpointed address accessed by the memory access or set of contiguous memory
//   accesses that triggered the watchpoint.

boolean AddressNotInNaturallyAlignedBlock(bits(64) address);
```

## Library pseudocode for shared/functions/system/BranchTargetCheck

```
// BranchTargetCheck()
// =====
// This function is executed checks if the current instruction is a valid target for a branch
// taken into, or inside, a guarded page. It is executed on every cycle once the current
// instruction has been decoded and the values of InGuardedPage and BTypeCompatible have been
// determined for the current instruction.

BranchTargetCheck()
    assert HaveBTIExt() && !UsingAArch32();

    // The branch target check considers two state variables:
    // * InGuardedPage, which is evaluated during instruction fetch.
    // * BTypeCompatible, which is evaluated during instruction decode.
    if InGuardedPage && PSTATE.BTYPE != '00' && !BTypeCompatible && !Halted() then
        bits(64) pc = ThisInstrAddr(64);
        AArch64.BranchTargetException(pc<51:0>);

    boolean branch_instr = AArch64.ExecutingBROrBLROrRetInstr();
    boolean bti_instr    = AArch64.ExecutingBTIInstr();

    // PSTATE.BTYPE defaults to 00 for instructions that do not explicitly set BTYPE.
    if !(branch_instr || bti_instr) then
        BTypeNext = '00';
```

## Library pseudocode for shared/functions/system/ClearEventRegister

```
// ClearEventRegister()
// =====
// Clear the Event Register of this PE.

ClearEventRegister()
    EventRegister = '0';
    return;
```

## Library pseudocode for shared/functions/system/ConditionHolds

```
// ConditionHolds()
// =====
// Return TRUE iff COND currently holds

boolean ConditionHolds(bits(4) cond)
    // Evaluate base condition.
    boolean result;
    case cond<3:1> of
        when '000' result = (PSTATE.Z == '1'); // EQ or NE
        when '001' result = (PSTATE.C == '1'); // CS or CC
        when '010' result = (PSTATE.N == '1'); // MI or PL
        when '011' result = (PSTATE.V == '1'); // VS or VC
        when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0'); // HI or LS
        when '101' result = (PSTATE.N == PSTATE.V); // GE or LT
        when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0'); // GT or LE
        when '111' result = TRUE; // AL

    // Condition flag values in the set '111x' indicate always true
    // Otherwise, invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;

    return result;
```

## Library pseudocode for shared/functions/system/ConsumptionOfSpeculativeDataBarrier

```
// ConsumptionOfSpeculativeDataBarrier()
// =====

ConsumptionOfSpeculativeDataBarrier();
```

## Library pseudocode for shared/functions/system/CurrentInstrSet

```
// CurrentInstrSet()
// =====

InstrSet CurrentInstrSet()
    InstrSet result;
    if UsingAArch32() then
        result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
    else
        result = InstrSet_A64;
    return result;
```

## Library pseudocode for shared/functions/system/CurrentPL

```
// CurrentPL()
// =====

PrivilegeLevel CurrentPL()
    return PLOfEL(PSTATE.EL);
```

## Library pseudocode for shared/functions/system/CurrentSecurityState

```
// CurrentSecurityState()
// =====
// Returns the effective security state at the exception level based off current settings.

SecurityState CurrentSecurityState()
    return SecurityStateAtEL(PSTATE.EL);
```

## Library pseudocode for shared/functions/system/DSBAlias

```
// DSBAlias
// =====
// Aliases of DSB.

enumeration DSBAlias {DSBAlias_SSBB, DSBAlias_PSSBB, DSBAlias_DSB};
```

## Library pseudocode for shared/functions/system/EL0

```
constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';
```

## Library pseudocode for shared/functions/system/EL2Enabled

```
// EL2Enabled()
// =====
// Returns TRUE if EL2 is present and executing
// - with the PE in Non-secure state when Non-secure EL2 is implemented, or
// - with the PE in Realm state when Realm EL2 is implemented, or
// - with the PE in Secure state when Secure EL2 is implemented and enabled, or
// - when EL3 is not implemented.

boolean EL2Enabled()
    return HaveEL(EL2) && (!HaveEL(EL3) || SCR_curr[].NS == '1' || IsSecureEL2Enabled());
```



## Library pseudocode for shared/functions/system/EL3SDDUndef

```
// EL3SDDUndef()
// =====
// Returns TRUE if in Debug state and EDSCR.SDD is set.

boolean EL3SDDUndef()
    return Halted() && EDSCR.SDD == '1';
```

## Library pseudocode for shared/functions/system/EL3SDDUndefPriority

```
// EL3SDDUndefPriority()
// =====
// Returns TRUE if in Debug state, EDSCR.SDD is set, and an EL3 trap by an
// EL3 control register has priority over other traps.
// The IMPLEMENTATION_DEFINED priority may be different for each case.

boolean EL3SDDUndefPriority()
    return (Halted() && EDSCR.SDD == '1' &&
        boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'");
```

## Library pseudocode for shared/functions/system/ELFromM32

```
// ELFromM32()
// =====

(boolean, bits(2)) ELFromM32(bits(5) mode)
    // Convert an AArch32 mode encoding to an Exception level.
    // Returns (valid, EL):
    //   'valid' is TRUE if 'mode<4:0>' encodes a mode that is both valid for this implementation
    //           and the current value of SCR.NS/SCR_EL3.NS.
    //   'EL' is the Exception level decoded from 'mode'.
    bits(2) el;
    boolean valid = !BadMode(mode); // Check for modes that are not valid for this implementation
    bits(2) effective_nse_ns = EffectiveSCR\_EL3\_NSE() : EffectiveSCR\_EL3\_NS();

    case mode of
        when M32\_Monitor
            el = EL3;
        when M32\_Hyp
            el = EL2;
        when M32\_FIQ, M32\_IRQ, M32\_Svc, M32\_Abort, M32\_Undef, M32\_System
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            el = (if HaveEL(EL3) && !HaveAArch64() && SCR.NS == '0' then EL3 else EL1);
        when M32\_User
            el = EL0;
        otherwise
            valid = FALSE; // Passed an illegal mode value

    if valid && el == EL2 && HaveEL(EL3) && SCR\_curr{}.NS == '0' then
        valid = FALSE; // EL2 only valid in Non-secure state in AArch32

    elsif valid && HaveRME() && effective_nse_ns == '10' then
        valid = FALSE; // Illegal Exception Return from EL3 if SCR_EL3.<NSE,NS>
        // selects a reserved encoding

    if !valid then el = bits(2) UNKNOWN;
    return (valid, el);
```

## Library pseudocode for shared/functions/system/ELFromSPSR

```
// ELFromSPSR()
// =====

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
// 'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
// 'EL' is the Exception level decoded from 'spsr'.

(boolean, bits(2)) ELFromSPSR(bits(N) spsr)
    bits(2) el;
    boolean valid;
    bits(2) effective_nse_ns;
    if spsr<4> == '0' then // AArch64 state
        el = spsr<3:2>;
        effective_nse_ns = EffectiveSCR\_EL3\_NSE\(\) : EffectiveSCR\_EL3\_NS\(\);
        if !HaveAArch64\(\) then
            valid = FALSE; // No AArch64 support
        elsif !HaveEL(el) then
            valid = FALSE; // Exception level not implemented
        elsif spsr<1> == '1' then
            valid = FALSE; // M[1] must be 0
        elsif el == EL0 && spsr<0> == '1' then
            valid = FALSE; // for EL0, M[0] must be 0
        elsif HaveRME\(\) && el != EL3 && effective_nse_ns == '10' then
            valid = FALSE; // Only EL3 valid in Root state
        elsif el == EL2 && HaveEL(EL3) && !IsSecureEL2Enabled\(\) && SCR_EL3.NS == '0' then
            valid = FALSE; // Unless Secure EL2 is enabled, EL2 valid only in Non-secure state
        else
            valid = TRUE;
    elsif HaveAArch32\(\) then // AArch32 state
        (valid, el) = ELFromM32(spsr<4:0>);
    else
        valid = FALSE;

    if !valid then el = bits(2) UNKNOWN;
    return (valid, el);
```

## Library pseudocode for shared/functions/system/ELIsInHost

```
// ELIsInHost()
// =====

boolean ELIsInHost(bits(2) el)
    if !HaveVirtHostExt\(\) || ELUsingAArch32(EL2) then
        return FALSE;
    case el of
        when EL3
            return FALSE;
        when EL2
            return EL2Enabled\(\) && HCR_EL2.E2H == '1';
        when EL1
            return FALSE;
        when EL0
            return EL2Enabled\(\) && HCR_EL2.<E2H,TGE> == '11';
    otherwise
        Unreachable\(\);
```

## Library pseudocode for shared/functions/system/ELStateUsingAArch32

```
// ELStateUsingAArch32()
// =====

boolean ELStateUsingAArch32(bits(2) el, boolean secure)
    // See ELStateUsingAArch32K() for description. Must only be called in circumstances where
    // result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
    (known, aarch32) = ELStateUsingAArch32K(el, secure);
    assert known;
    return aarch32;
```

## Library pseudocode for shared/functions/system/ELStateUsingAArch32K

```
// ELStateUsingAArch32K()
// =====

(boolean,boolean) ELStateUsingAArch32K(bits(2) el, boolean secure)
    // Returns (known, aarch32):
    // 'known' is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
    // using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
    // 'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.
    if !HaveAArch32EL(el) then
        return (TRUE, FALSE); // Exception level is using AArch64
    elsif secure && el == EL2 then
        return (TRUE, FALSE); // Secure EL2 is using AArch64
    elsif !HaveAArch64() then
        return (TRUE, TRUE); // Highest Exception level, therefore all levels are using AArch32

    // Remainder of function deals with the interprocessing cases when highest
    // Exception level is using AArch64

    boolean aarch32 = boolean UNKNOWN;
    boolean known = TRUE;

    aarch32_below_el3 = (HaveEL(EL3) && (!secure || !HaveSecureEL2Ext() || SCR_EL3.EEL2 == '0') &&
        SCR_EL3.RW == '0');
    aarch32_at_el1 = (aarch32_below_el3 ||
        (HaveEL(EL2) && (!secure || (HaveSecureEL2Ext() && SCR_EL3.EEL2 == '1')) &&
        !(HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11') &&
        HCR_EL2.RW == '0'));
    if el == EL0 && !aarch32_at_el1 then // Only know if EL0 using AArch32 from PSTATE
        if PSTATE.EL == EL0 then
            aarch32 = PSTATE.nRW == '1'; // EL0 controlled by PSTATE
        else
            known = FALSE; // EL0 state is UNKNOWN
    else
        aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL1,EL0});

    if !known then aarch32 = boolean UNKNOWN;
    return (known, aarch32);
```

## Library pseudocode for shared/functions/system/ELUsingAArch32

```
// ELUsingAArch32()
// =====

boolean ELUsingAArch32(bits(2) el)
    return ELStateUsingAArch32(el, IsSecureBelowEL3());
```

## Library pseudocode for shared/functions/system/ELUsingAArch32K

```
// ELUsingAArch32K()
// =====

(boolean,boolean) ELUsingAArch32K(bits(2) el)
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());
```

### Library pseudocode for shared/functions/system/EffectiveEA

```
// EffectiveEA()
// =====
// Returns effective SCR_EL3.EA value

bit EffectiveEA()
    if Halted\(\) && EDSCR.SDD == '0' then
        return '0';
    else
        return if HaveAArch64\(\) then SCR_EL3.EA else SCR.EA;
```

### Library pseudocode for shared/functions/system/EffectiveSCR\_EL3\_NS

```
// EffectiveSCR_EL3_NS()
// =====
// Return Effective SCR_EL3.NS value.

bit EffectiveSCR_EL3_NS()
    if !HaveSecureState\(\) then
        return '1';
    elsif !HaveEL\(EL3\) then
        return '0';
    else
        return SCR_EL3.NS;
```

### Library pseudocode for shared/functions/system/EffectiveSCR\_EL3\_NSE

```
// EffectiveSCR_EL3_NSE()
// =====
// Return Effective SCR_EL3.NSE value.

bit EffectiveSCR_EL3_NSE()
    return if !HaveRME\(\) then '0' else SCR_EL3.NSE;
```

### Library pseudocode for shared/functions/system/EffectiveSCR\_EL3\_RW

```
// EffectiveSCR_EL3_RW()
// =====
// Returns effective SCR_EL3.RW value

bit EffectiveSCR_EL3_RW()
    if !HaveAArch64\(\) then
        return '0';
    if !HaveAArch32EL\(EL2\) && !HaveAArch32EL\(EL1\) then
        return '1';
    if HaveAArch32EL\(EL1\) then
        if !HaveAArch32EL\(EL2\) && SCR_EL3.NS == '1' then
            return '1';
        if HaveSecureEL2Ext\(\) && SCR_EL3.EEL2 == '1' && SCR_EL3.NS == '0' then
            return '1';
    return SCR_EL3.RW;
```

### Library pseudocode for shared/functions/system/EffectiveTGE

```
// EffectiveTGE()
// =====
// Returns effective TGE value

bit EffectiveTGE()
    if EL2Enabled\(\) then
        return if ELUsingAArch32\(EL2\) then HCR.TGE else HCR_EL2.TGE;
    else
        return '0';           // Effective value of TGE is zero
```

### Library pseudocode for shared/functions/system/EndOfInstruction

```
// EndOfInstruction()
// =====
// Terminate processing of the current instruction.

EndOfInstruction();
```

### Library pseudocode for shared/functions/system/EnterLowPowerState

```
// EnterLowPowerState()
// =====
// PE enters a low-power state.

EnterLowPowerState();
```

### Library pseudocode for shared/functions/system/EventRegister

```
bits(1) EventRegister;
```

### Library pseudocode for shared/functions/system/ExceptionalOccurrenceTargetState

```
// ExceptionalOccurrenceTargetState
// =====
// Enumeration to represent the target state of an Exceptional Occurrence.
// The Exceptional Occurrence can be either Exception or Debug State entry.

enumeration ExceptionalOccurrenceTargetState {
    AArch32_NonDebugState,
    AArch64_NonDebugState,
    DebugState
};
```

### Library pseudocode for shared/functions/system/FIQPending

```
// FIQPending()
// =====
// Returns a tuple indicating if there is any pending physical FIQ
// and if the pending FIQ has superpriority.

(boolean, boolean) FIQPending();
```

### Library pseudocode for shared/functions/system/GetAccumulatedFPExceptions

```
// GetAccumulatedFPExceptions()
// =====
// Returns FP exceptions accumulated by the PE.

bits(8) GetAccumulatedFPExceptions();
```

### Library pseudocode for shared/functions/system/GetLoadStoreType

```
// GetLoadStoreType()
// =====
// Returns the Load/Store Type. Used when a Translation fault,
// Access flag fault, or Permission fault generates a Data Abort.

bits(2) GetLoadStoreType();
```

## Library pseudocode for shared/functions/system/GetPSRFromPSTATE

```
// GetPSRFromPSTATE()
// =====
// Return a PSR value which represents the current PSTATE

bits(N) GetPSRFromPSTATE(ExceptionalOccurrenceTargetState targetELState, integer N)
    if UsingAArch32() && targetELState == AArch32\_NonDebugState then
        assert N == 32;
    else
        assert N == 64;

    bits(N) spsr = Zeros(N);
    spsr<31:28> = PSTATE.<N,Z,C,V>;
    if HavePANExt() then spsr<22> = PSTATE.PAN;
    spsr<20> = PSTATE.IL;
    if PSTATE.nRW == '1' then // AArch32 state
        if targetELState != AArch32\_NonDebugState then
            spsr<33> = PSTATE.PPEND;
        spsr<27> = PSTATE.Q;
        spsr<26:25> = PSTATE.IT<1:0>;
        if HaveSSBSExt() then spsr<23> = PSTATE.SSBS;
        if HaveDITEExt() then
            if targetELState == AArch32\_NonDebugState then
                spsr<21> = PSTATE.DIT;
            else // AArch64_NonDebugState or DebugState
                spsr<24> = PSTATE.DIT;
        if targetELState IN {AArch64\_NonDebugState, DebugState} then
            spsr<21> = PSTATE.SS;
        spsr<19:16> = PSTATE.GE;
        spsr<15:10> = PSTATE.IT<7:2>;
        spsr<9> = PSTATE.E;
        spsr<8:6> = PSTATE.<A,I,F>; // No PSTATE.D in AArch32 state
        spsr<5> = PSTATE.T;
        assert PSTATE.M<4> == PSTATE.nRW; // bit [4] is the discriminator
        spsr<4:0> = PSTATE.M;
    else // AArch64 state
        if HaveGCS() then spsr<34> = PSTATE.EXLOCK;
        if IsFeatureImplemented(FEAT_SEBEP) then spsr<33> = PSTATE.PPEND;
        if IsFeatureImplemented(FEAT_EBEP) then spsr<32> = PSTATE.PM;
        if HaveMTEEExt() then spsr<25> = PSTATE.TCO;
        if HaveDITEExt() then spsr<24> = PSTATE.DIT;
        if HaveUAOExt() then spsr<23> = PSTATE.UAO;
        spsr<21> = PSTATE.SS;
        if HaveFeatNMI() then spsr<13> = PSTATE.ALLINT;
        if HaveSSBSExt() then spsr<12> = PSTATE.SSBS;
        if HaveBTIExt() then spsr<11:10> = PSTATE.BTYPE;
        spsr<9:6> = PSTATE.<D,A,I,F>;
        spsr<4> = PSTATE.nRW;
        spsr<3:2> = PSTATE.EL;
        spsr<0> = PSTATE.SP;
    return spsr;
```

## Library pseudocode for shared/functions/system/HasArchVersion

```
// HasArchVersion()
// =====
// Returns TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(boolean version)
    return version;
```

### Library pseudocode for shared/functions/system/HaveAArch32

```
// HaveAArch32()
// =====
// Return TRUE if AArch32 state is supported at at least EL0.

boolean HaveAArch32()
    return IsFeatureImplemented(FEAT_AA32EL0);
```

### Library pseudocode for shared/functions/system/HaveAArch32EL

```
// HaveAArch32EL()
// =====
// Return TRUE if Exception level 'el' supports AArch32 in this implementation

boolean HaveAArch32EL(bits(2) el)
    case el of
        when EL0 return IsFeatureImplemented(FEAT_AA32EL0);
        when EL1 return IsFeatureImplemented(FEAT_AA32EL1);
        when EL2 return IsFeatureImplemented(FEAT_AA32EL2);
        when EL3 return IsFeatureImplemented(FEAT_AA32EL3);
```

### Library pseudocode for shared/functions/system/HaveAArch64

```
// HaveAArch64()
// =====
// Return TRUE if the highest Exception level is using AArch64 state.

boolean HaveAArch64()
    return (IsFeatureImplemented(FEAT_AA64EL0) || IsFeatureImplemented(FEAT_AA64EL1) ||
            IsFeatureImplemented(FEAT_AA64EL2) || IsFeatureImplemented(FEAT_AA64EL3));
```

### Library pseudocode for shared/functions/system/HaveEL

```
// HaveEL()
// =====
// Return TRUE if Exception level 'el' is supported

boolean HaveEL(bits(2) el)
    case el of
        when EL1, EL0
            return TRUE; // EL1 and EL0 must exist
        when EL2
            return IsFeatureImplemented(FEAT_AA64EL2) || IsFeatureImplemented(FEAT_AA32EL2);
        when EL3
            return IsFeatureImplemented(FEAT_AA64EL3) || IsFeatureImplemented(FEAT_AA32EL3);
        otherwise
            Unreachable();
```

## Library pseudocode for shared/functions/system/HaveELUsingSecurityState

```
// HaveELUsingSecurityState()
// =====
// Returns TRUE if Exception level 'el' with Security state 'secure' is supported,
// FALSE otherwise.

boolean HaveELUsingSecurityState(bits(2) el, boolean secure)

    case el of
        when EL3
            assert secure;
            return HaveEL\(EL3\);
        when EL2
            if secure then
                return HaveEL\(EL2\) && HaveSecureEL2Ext\(\);
            else
                return HaveEL\(EL2\);
        otherwise
            return (HaveEL\(EL3\) ||
                (secure == boolean IMPLEMENTATION_DEFINED "Secure-only implementation"));
```

## Library pseudocode for shared/functions/system/HaveFP16Ext

```
// HaveFP16Ext()
// =====
// Return TRUE if FP16 extension is supported

boolean HaveFP16Ext()
    return IsFeatureImplemented(FEAT_FP16);
```

## Library pseudocode for shared/functions/system/HaveSecureState

```
// HaveSecureState()
// =====
// Return TRUE if Secure State is supported.

boolean HaveSecureState()
    if !HaveEL\(EL3\) then
        return SecureOnlyImplementation\(\);
    if HaveRME\(\) && !HaveSecureEL2Ext\(\) then
        return FALSE;
    return TRUE;
```

## Library pseudocode for shared/functions/system/HighestEL

```
// HighestEL()
// =====
// Returns the highest implemented Exception level.

bits(2) HighestEL()
    if HaveEL\(EL3\) then
        return EL3;
    elsif HaveEL\(EL2\) then
        return EL2;
    else
        return EL1;
```

## Library pseudocode for shared/functions/system/Hint\_CLRBHB

```
// Hint_CLRBHB()
// =====
// Provides a hint to clear the branch history for the current context.

Hint_CLRBHB();
```



## Library pseudocode for shared/functions/system/Hint\_DGH

```
// Hint_DGH()
// =====
// Provides a hint to close any gathering occurring within the micro-architecture.

Hint_DGH();
```

## Library pseudocode for shared/functions/system/Hint\_WFE

```
// Hint_WFE()
// =====
// Provides a hint indicating that the PE can enter a low-power state
// and remain there until a wakeup event occurs or, for WFET, a local
// timeout event is generated when the virtual timer value equals or
// exceeds the supplied threshold value.

Hint_WFE(integer localtimeout, WfxType wfxtype)
    if IsEventRegisterSet() then
        ClearEventRegister();
    elseif HaveFeatWfxT() && LocalTimeoutEvent(localtimeout) then
        // No further operation if the local timeout has expired.
        EndOfInstruction();
    else
        bits(2) target_el;
        trap = FALSE;
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS which may be EL1 or EL2.
            if HaveTWEDEExt() then
                sctlr = SCTLR\_ELx[];
                trap = sctlr.nTWE == '0';
                target_el = EL1;
            else
                AArch64.CheckForWfxTrap(EL1, wfxtype);
        if !trap && PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            if HaveTWEDEExt() then
                trap = HCR_EL2.TWE == '1';
                target_el = EL2;
            else
                AArch64.CheckForWfxTrap(EL2, wfxtype);

        if !trap && HaveEL(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            if HaveTWEDEExt() then
                trap = SCR_EL3.TWE == '1';
                target_el = EL3;
            else
                AArch64.CheckForWfxTrap(EL3, wfxtype);

        if trap && PSTATE.EL != EL3 then
            // Determine if trap delay is enabled and delay amount
            (delay_enabled, delay) = WFETrapDelay(target_el);
            if !WaitForEventUntilDelay(delay_enabled, delay) then
                // Event did not arrive before delay expired so trap WFE
                AArch64.WfxTrap(wfxtype, target_el);
        else
            WaitForEvent(localtimeout);
```

## Library pseudocode for shared/functions/system/Hint\_WFI

```
// Hint_WFI()
// =====
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs or, for WFIT, a local timeout
// event is generated when the virtual timer value equals or exceeds the
// supplied threshold value.

Hint_WFI(integer localtimeout, WfxType wfxtype)
    if HaveTME() && TSTATE.depth > 0 then
        FailTransaction(TMFailure_ERR, FALSE);

    if InterruptPending() || (HaveFeatWfxT() && LocalTimeoutEvent(localtimeout)) then
        // No further operation if an interrupt is pending or the local timeout has expired.
        EndOfInstruction();
    else
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS.
            AArch64.CheckForWfxTrap(EL1, wfxtype);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWfxTrap(EL2, wfxtype);
        if HaveEL(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWfxTrap(EL3, wfxtype);
        WaitForInterrupt(localtimeout);
```

## Library pseudocode for shared/functions/system/Hint\_Yield

```
// Hint_Yield()
// =====
// Provides a hint that the task performed by a thread is of low
// importance so that it could yield to improve overall performance.

Hint_Yield();
```

## Library pseudocode for shared/functions/system/IRQPending

```
// IRQPending()
// =====
// Returns a tuple indicating if there is any pending physical IRQ
// and if the pending IRQ has superpriority.

(boolean, boolean) IRQPending();
```

## Library pseudocode for shared/functions/system/IllegalExceptionReturn

```
// IllegalExceptionReturn()
// =====

boolean IllegalExceptionReturn(bits(N) spsr)

    // Check for illegal return:
    // * To an unimplemented Exception level.
    // * To EL2 in Secure state, when SecureEL2 is not enabled.
    // * To EL0 using AArch64 state, with SPSR.M[0]==1.
    // * To AArch64 state with SPSR.M[1]==1.
    // * To AArch32 state with an illegal value of SPSR.M.
    (valid, target) = ELFromSPSR(spsr);
    if !valid then return TRUE;

    // Check for return to higher Exception level
    if UInt(target) > UInt(PSTATE.EL) then return TRUE;

    spsr_mode_is_aarch32 = (spsr<4> == '1');

    // Check for illegal return:
    // * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    //   Execution state used in the Exception level being returned to, as determined by
    //   the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
    // * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
    //   SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
    // * To AArch64 state from AArch32 state (should be caught by above)
    (known, target_el_is_aarch32) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !ELUsingAArch32(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

    // Check for illegal return from AArch32 to AArch64
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;

    // Check for illegal return to EL1 when HCR.TGE is set and when either of
    // * SecureEL2 is enabled.
    // * SecureEL2 is not enabled and EL1 is in Non-secure state.
    if EL2Enabled() && target == EL1 && HCR_EL2.TGE == '1' then
        if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE;

    if (HaveGCS() && PSTATE.EXLOCK == '0' &&
        PSTATE.EL == target && GetCurrentEXLOCKEN()) then
        return TRUE;

    return FALSE;
```

## Library pseudocode for shared/functions/system/InstrSet

```
// InstrSet
// =====

enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

## Library pseudocode for shared/functions/system/InstructionSynchronizationBarrier

```
// InstructionSynchronizationBarrier()
// =====
InstructionSynchronizationBarrier();
```

### Library pseudocode for shared/functions/system/InterruptPending

```
// InterruptPending()
// =====
// Returns TRUE if there are any pending physical or virtual
// interrupts, and FALSE otherwise.

boolean InterruptPending()
    boolean pending_virtual_interrupt = FALSE;
    (irq_pending, -) = IRQPending\(\);
    (fiq_pending, -) = FIQPending\(\);
    boolean pending_physical_interrupt = (irq_pending || fiq_pending ||
                                         IsPhysicalErrorPending\(\));

    if EL2Enabled\(\) && PSTATE.EL IN {EL0, EL1} && HCR_EL2.TGE == '0' then
        boolean virq_pending = HCR_EL2.IMO == '1' && (VirtualIRQPending\(\) || HCR_EL2.VI == '1') ;
        boolean vfiq_pending = HCR_EL2.FMO == '1' && (VirtualFIQPending\(\) || HCR_EL2.VF == '1');
        boolean vsei_pending = HCR_EL2.AMO == '1' && (IsVirtualErrorPending\(\) ||
                                                    HCR_EL2.VSE == '1');

        pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending;

    return pending_physical_interrupt || pending_virtual_interrupt;
```

### Library pseudocode for shared/functions/system/IsASEInstruction

```
// IsASEInstruction()
// =====
// Returns TRUE if the current instruction is an ASIMD or SVE vector instruction.

boolean IsASEInstruction();
```

### Library pseudocode for shared/functions/system/IsCurrentSecurityState

```
// IsCurrentSecurityState()
// =====
// Returns TRUE if the current Security state matches
// the given Security state, and FALSE otherwise.

boolean IsCurrentSecurityState(SecurityState ss)
    return CurrentSecurityState\(\) == ss;
```

### Library pseudocode for shared/functions/system/IsEventRegisterSet

```
// IsEventRegisterSet()
// =====
// Return TRUE if the Event Register of this PE is set, and FALSE if it is clear.

boolean IsEventRegisterSet()
    return EventRegister == '1';
```

### Library pseudocode for shared/functions/system/IsHighestEL

```
// IsHighestEL()
// =====
// Returns TRUE if given exception level is the highest exception level implemented

boolean IsHighestEL(bits(2) el)
    return HighestEL\(\) == el;
```

### Library pseudocode for shared/functions/system/IsInHost

```
// IsInHost()
// =====

boolean IsInHost()
    return ELIsInHost(PSTATE.EL);
```

## Library pseudocode for shared/functions/system/IsSecure

```
// IsSecure()
// =====
// Returns TRUE if current Exception level is in Secure state.

boolean IsSecure()
    if HaveEL(EL3) && !UsingAArch32() && PSTATE.EL == EL3 then
        return TRUE;
    elseif HaveEL(EL3) && UsingAArch32() && PSTATE.M == M32_Monitor then
        return TRUE;
    return IsSecureBelowEL3();
```

## Library pseudocode for shared/functions/system/IsSecureBelowEL3

```
// IsSecureBelowEL3()
// =====
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// Differs from IsSecure in that it ignores the current EL or Mode
// in considering security state.
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

boolean IsSecureBelowEL3()
    if HaveEL(EL3) then
        return SCR_curr[].NS == '0';
    elseif HaveEL(EL2) && (!HaveSecureEL2Ext() || !HaveAArch64()) then
        // If Secure EL2 is not an architecture option then we must be Non-secure.
        return FALSE;
    else
        // TRUE if processor is Secure or FALSE if Non-secure.
        return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

## Library pseudocode for shared/functions/system/IsSecureEL2Enabled

```
// IsSecureEL2Enabled()
// =====
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.

boolean IsSecureEL2Enabled()
    if HaveEL(EL2) && HaveSecureEL2Ext() then
        if HaveEL(EL3) then
            if !ELUsingAArch32(EL3) && SCR_EL3.EEL2 == '1' then
                return TRUE;
            else
                return FALSE;
        else
            return SecureOnlyImplementation();
    else
        return FALSE;
```

## Library pseudocode for shared/functions/system/LocalTimeoutEvent

```
// LocalTimeoutEvent()
// =====
// Returns TRUE if CNTVCT_EL0 equals or exceeds the localtimeout value.

boolean LocalTimeoutEvent(integer localtimeout);
```

### Library pseudocode for shared/functions/system/Mode\_Bits

```
constant bits(5) M32_User      = '10000';
constant bits(5) M32_FIQ      = '10001';
constant bits(5) M32_IRQ      = '10010';
constant bits(5) M32_Svc      = '10011';
constant bits(5) M32_Monitor  = '10110';
constant bits(5) M32_Abort    = '10111';
constant bits(5) M32_Hyp      = '11010';
constant bits(5) M32_Undef    = '11011';
constant bits(5) M32_System   = '11111';
```

### Library pseudocode for shared/functions/system/NonSecureOnlyImplementation

```
// NonSecureOnlyImplementation()
// =====
// Returns TRUE if the security state is always Non-secure for this implementation.

boolean NonSecureOnlyImplementation()
    return boolean IMPLEMENTATION_DEFINED "Non-secure only implementation";
```

### Library pseudocode for shared/functions/system/PLOfEL

```
// PLOfEL()
// =====

PrivilegeLevel PLOfEL(bits(2) el)
    case el of
        when EL3    return if !HaveAArch64() then PL1 else PL3;
        when EL2    return PL2;
        when EL1    return PL1;
        when EL0    return PL0;
```

### Library pseudocode for shared/functions/system/PSTATE

```
ProcState PSTATE;
```

### Library pseudocode for shared/functions/system/PhysicalCountInt

```
// PhysicalCountInt()
// =====
// Returns the integral part of physical count value of the System counter.

bits(64) PhysicalCountInt()
    return PhysicalCount<87:24>;
```

### Library pseudocode for shared/functions/system/PrivilegeLevel

```
// PrivilegeLevel
// =====
// Privilege Level abstraction.

enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};
```

## Library pseudocode for shared/functions/system/ProcState

```
// ProcState
// =====
// Armv8 processor state bits.
// There is no significance to the field order.

type ProcState is (
  bits (1) N,          // Negative condition flag
  bits (1) Z,          // Zero condition flag
  bits (1) C,          // Carry condition flag
  bits (1) V,          // Overflow condition flag
  bits (1) D,          // Debug mask bit [AArch64 only]
  bits (1) A,          // SError interrupt mask bit
  bits (1) I,          // IRQ mask bit
  bits (1) F,          // FIQ mask bit
  bits (1) EXLOCK,     // Lock exception return state
  bits (1) PAN,        // Privileged Access Never Bit [v8.1]
  bits (1) UAO,        // User Access Override [v8.2]
  bits (1) DIT,        // Data Independent Timing [v8.4]
  bits (1) TCO,        // Tag Check Override [v8.5, AArch64 only]
  bits (1) PM,        // PMU exception Mask
  bits (1) PPEND,      // synchronous PMU exception to be observed
  bits (2) BTYPE,     // Branch Type [v8.5]
  bits (1) ZA,        // Accumulation array enabled [SME]
  bits (1) SM,        // Streaming SVE mode enabled [SME]
  bits (1) ALLINT,     // Interrupt mask bit
  bits (1) SS,        // Software step bit
  bits (1) IL,        // Illegal Execution state bit
  bits (2) EL,        // Exception level
  bits (1) nRW,       // Execution state: 0=AArch64, 1=AArch32
  bits (1) SP,        // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
  bits (1) Q,         // Cumulative saturation flag [AArch32 only]
  bits (4) GE,        // Greater than or Equal flags [AArch32 only]
  bits (1) SSBS,      // Speculative Store Bypass Safe
  bits (8) IT,        // If-then bits, RES0 in CPSR [AArch32 only]
  bits (1) J,         // J bit, RES0 [AArch32 only, RES0 in SPSR and CPSR]
  bits (1) T,        // T32 bit, RES0 in CPSR [AArch32 only]
  bits (1) E,        // Endianness bit [AArch32 only]
  bits (5) M         // Mode field [AArch32 only]
)
```

## Library pseudocode for shared/functions/system/RestoredITBits

```
// RestoredITBits()
// =====
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(N) spsr)
  it = spsr<15:10,26:25>;

  // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
  // to zero or copied from the SPSR.
  if PSTATE.IL == '1' then
    if ConstrainUnpredictableBool(Unpredictable\_ILZEROIT) then return '00000000';
    else return it;

  // The IT bits are forced to zero when they are set to a reserved value.
  if !IsZero(it<7:4>) && IsZero(it<3:0>) then
    return '00000000';

  // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
  // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
  itd = if PSTATE.EL == EL2 then HSCTLR.ITD else SCTLR.ITD;
  if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then
    return '00000000';
  else
    return it;
```

### Library pseudocode for shared/functions/system/SCRType

```
type SCRType;
```

### Library pseudocode for shared/functions/system/SCR\_curr

```
// SCR_curr[]
// =====

SCRType SCR_curr[]
    // AArch32 secure & AArch64 EL3 registers are not architecturally mapped
    assert HaveEL(EL3);
    bits(64) r;
    if !HaveAArch64() then
        r = ZeroExtend(SCR, 64);
    else
        r = SCR_EL3;
    return r;
```

### Library pseudocode for shared/functions/system/SecureOnlyImplementation

```
// SecureOnlyImplementation()
// =====
// Returns TRUE if the security state is always Secure for this implementation.

boolean SecureOnlyImplementation()
    return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

### Library pseudocode for shared/functions/system/SecurityState

```
// SecurityState
// =====
// The Security state of an execution context

enumeration SecurityState {
    SS_NonSecure,
    SS_Root,
    SS_Realm,
    SS_Secure
};
```



## Library pseudocode for shared/functions/system/SecurityStateAtEL

```
// SecurityStateAtEL()
// =====
// Returns the effective security state at the exception level based off current settings.

SecurityState SecurityStateAtEL(bits(2) EL)
    if HaveRME\(\) then
        if EL == EL3 then return SS\_Root;
        effective_nse_ns = SCR_EL3.NSE : EffectiveSCR\_EL3\_NS\(\);
        case effective_nse_ns of
            when '00' if HaveSecureEL2Ext\(\) then return SS\_Secure; else Unreachable\(\);
            when '01' return SS\_NonSecure;
            when '11' return SS\_Realm;
            otherwise Unreachable\(\);

    if !HaveEL\(EL3\) then
        if SecureOnlyImplementation\(\) then
            return SS\_Secure;
        else
            return SS\_NonSecure;
    elsif EL == EL3 then
        return SS\_Secure;
    else
        // For EL2 call only when EL2 is enabled in current security state
        assert(EL != EL2 || EL2Enabled\(\));
        if !ELUsingAArch32\(EL3\) then
            return if SCR_EL3.NS == '1' then SS\_NonSecure else SS\_Secure;
        else
            return if SCR.NS == '1' then SS\_NonSecure else SS\_Secure;
```

## Library pseudocode for shared/functions/system/SendEvent

```
// SendEvent()
// =====
// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed.

SendEvent();
```

## Library pseudocode for shared/functions/system/SendEventLocal

```
// SendEventLocal()
// =====
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to be executed.

SendEventLocal()
    EventRegister = '1';
    return;
```

## Library pseudocode for shared/functions/system/SetAccumulatedFPExceptions

```
// SetAccumulatedFPExceptions()
// =====
// Stores FP Exceptions accumulated by the PE.

SetAccumulatedFPExceptions(bits(8) accumulated_exceptions);
```

## Library pseudocode for shared/functions/system/SetPSTATEFromPSR

```
// SetPSTATEFromPSR()
// =====

SetPSTATEFromPSR(bits(N) spsr)
    boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    SetPSTATEFromPSR(spsr, illegal_psr_state);

// SetPSTATEFromPSR()
// =====
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(N) spsr_in, boolean illegal_psr_state)
    bits(N) spsr = spsr_in;
    boolean from_aarch64 = !UsingAArch32();
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    if IsFeatureImplemented(FEAT_SEBEP) then
        assert N == 64;
        ExceptionReturnPPEND(ZeroExtend(spsr, 64));

    ShouldAdvanceSS = FALSE;
    if illegal_psr_state then
        PSTATE.IL = '1';
        if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
        if HaveBTIExt() then PSTATE.BTYPE = bits(2) UNKNOWN;
        if HaveUAOExt() then PSTATE.UAO = bit UNKNOWN;
        if HaveDITExt() then PSTATE.DIT = bit UNKNOWN;
        if HaveMTEEExt() then PSTATE.TCO = bit UNKNOWN;
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if spsr<4> == '1' then // AArch32 state
            AArch32.WriteMode(spsr<4:0>; // Sets PSTATE.EL correctly
            if HaveSSBSExt() then PSTATE.SSBS = spsr<23>;
        else // AArch64 state
            PSTATE.nRW = '0';
            PSTATE.EL = spsr<3:2>;
            PSTATE.SP = spsr<0>;
            if HaveBTIExt() then PSTATE.BTYPE = spsr<11:10>;
            if HaveSSBSExt() then PSTATE.SSBS = spsr<12>;
            if HaveUAOExt() then PSTATE.UAO = spsr<23>;
            if HaveDITExt() then PSTATE.DIT = spsr<24>;
            if HaveMTEEExt() then PSTATE.TCO = spsr<25>;
            if HaveGCS() then PSTATE.EXLOCK = spsr<34>;

        // If PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the T bit is set to zero or
        // copied from SPSR.
        if PSTATE.IL == '1' && PSTATE.nRW == '1' then
            if ConstrainUnpredictableBool(Unpredictable\_ILZEROT) then spsr<5> = '0';

        // State that is reinstated regardless of illegal exception return
        PSTATE.<N,Z,C,V> = spsr<31:28>;
        if HavePANExt() then PSTATE.PAN = spsr<22>;
        if PSTATE.nRW == '1' then // AArch32 state
            PSTATE.Q = spsr<27>;
            PSTATE.IT = RestoredITBits(spsr);
            ShouldAdvanceIT = FALSE;
            if HaveDITExt() then
                PSTATE.DIT = (if (Restarting() || from_aarch64) then spsr<24> else spsr<21>);
            PSTATE.GE = spsr<19:16>;
            PSTATE.E = spsr<9>;
            PSTATE.<A,I,F> = spsr<8:6>; // No PSTATE.D in AArch32 state
            PSTATE.T = spsr<5>; // PSTATE.J is RES0
        else // AArch64 state
            PSTATE.PM = spsr<32>;
            if HaveFeatNMI() then PSTATE.ALLINT = spsr<13>;
            PSTATE.<D,A,I,F> = spsr<9:6>; // No PSTATE.<Q,IT,GE,E,T> in AArch64 state
        return;
```

### Library pseudocode for shared/functions/system/ShouldAdvanceHS

```
boolean ShouldAdvanceHS;
```

### Library pseudocode for shared/functions/system/ShouldAdvanceIT

```
boolean ShouldAdvanceIT;
```

### Library pseudocode for shared/functions/system/ShouldAdvanceSS

```
boolean ShouldAdvanceSS;
```

### Library pseudocode for shared/functions/system/ShouldSetPPEND

```
boolean ShouldSetPPEND;
```

### Library pseudocode for shared/functions/system/SmallestTranslationGranule

```
// SmallestTranslationGranule()
// =====
// Smallest implemented translation granule.

integer SmallestTranslationGranule()
    if boolean IMPLEMENTATION_DEFINED "Has 4K Translation Granule" then return 12;
    if boolean IMPLEMENTATION_DEFINED "Has 16K Translation Granule" then return 14;
    if boolean IMPLEMENTATION_DEFINED "Has 64K Translation Granule" then return 16;
```

### Library pseudocode for shared/functions/system/SpeculationBarrier

```
// SpeculationBarrier()
// =====

SpeculationBarrier();
```

### Library pseudocode for shared/functions/system/SyncCounterOverflowed

```
boolean SyncCounterOverflowed;
```

### Library pseudocode for shared/functions/system/SynchronizeContext

```
// SynchronizeContext()
// =====

SynchronizeContext();
```

### Library pseudocode for shared/functions/system/SynchronizeErrors

```
// SynchronizeErrors()
// =====
// Implements the error synchronization event.

SynchronizeErrors();
```

### Library pseudocode for shared/functions/system/TakeUnmaskedPhysicalSErrorInterrupts

```
// TakeUnmaskedPhysicalSErrorInterrupts()
// =====
// Take any pending unmasked physical SError interrupt.

TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);
```

### Library pseudocode for shared/functions/system/TakeUnmaskedSErrorInterrupts

```
// TakeUnmaskedSErrorInterrupts()
// =====
// Take any pending unmasked physical SError interrupt or unmasked virtual SError
// interrupt.

TakeUnmaskedSErrorInterrupts();
```

### Library pseudocode for shared/functions/system/ThisInstr

```
// ThisInstr()
// =====

bits(32) ThisInstr();
```

### Library pseudocode for shared/functions/system/ThisInstrLength

```
// ThisInstrLength()
// =====

integer ThisInstrLength();
```

### Library pseudocode for shared/functions/system/Unreachable

```
// Unreachable()
// =====

Unreachable()
    assert FALSE;
```

### Library pseudocode for shared/functions/system/UsingAArch32

```
// UsingAArch32()
// =====
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.

boolean UsingAArch32()
    boolean aarch32 = (PSTATE.nRW == '1');
    if !HaveAArch32() then assert !aarch32;
    if !HaveAArch64() then assert aarch32;
    return aarch32;
```

### Library pseudocode for shared/functions/system/ValidSecurityStateAtEL

```
// ValidSecurityStateAtEL()
// =====
// Returns TRUE if the current settings and architecture choices for this
// implementation permit a valid Security state at the indicated EL.

boolean ValidSecurityStateAtEL(bits(2) el)
    if !HaveEL(el) then
        return FALSE;

    if el == EL3 then
        return TRUE;

    if HaveRME() then
        bits(2) effective_nse_ns = SCR_EL3.NSE : EffectiveSCR_EL3_NS();
        if effective_nse_ns == '10' then
            return FALSE;

    if el == EL2 then
        return EL2Enabled();

    return TRUE;
```

### Library pseudocode for shared/functions/system/VirtualFIQPending

```
// VirtualFIQPending()
// =====
// Returns TRUE if there is any pending virtual FIQ.

boolean VirtualFIQPending();
```

### Library pseudocode for shared/functions/system/VirtualIRQPending

```
// VirtualIRQPending()
// =====
// Returns TRUE if there is any pending virtual IRQ.

boolean VirtualIRQPending();
```

### Library pseudocode for shared/functions/system/WFxType

```
// WFxType
// =====
// WFx instruction types.

enumeration WFxType {WfxType_WFE, WfxType_WFI, WfxType_WFET, WfxType_WFIT};
```

### Library pseudocode for shared/functions/system/WaitForEvent

```
// WaitForEvent()
// =====
// PE optionally suspends execution until one of the following occurs:
// - A WFE wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Event with Timeout (WFET) is executing, and a local timeout event occurs
// It is IMPLEMENTATION DEFINED whether restarting execution after the period of
// suspension causes the Event Register to be cleared.

WaitForEvent(integer localtimeout)
    if !(IsEventRegisterSet() || (HaveFeatWfxT() && LocalTimeoutEvent(localtimeout))) then
        EnterLowPowerState();
    return;
```

### Library pseudocode for shared/functions/system/WaitForInterrupt

```
// WaitForInterrupt()
// =====
// PE optionally suspends execution until one of the following occurs:
// - A WFI wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Interrupt with Timeout (WFIT) is executing, and a local timeout event occurs.

WaitForInterrupt(integer localtimeout)
    if !(HaveFeatWfxT() && LocalTimeoutEvent(localtimeout)) then
        EnterLowPowerState();
    return;
```

## Library pseudocode for shared/functions/system/WatchpointRelatedSyndrome

```
// WatchpointRelatedSyndrome()
// =====
// Update common Watchpoint related fields.

bits(24) WatchpointRelatedSyndrome(FaultRecord fault, bits(64) vaddress)
    bits(24) syndrome = Zeros(24);

    if fault.maybe_false_match then
        syndrome<16> = '1';
    else
        syndrome<16> = bit IMPLEMENTATION_DEFINED "WPF value on TRUE Watchpoint match";

    if IsSVEAccess(fault.accessdesc) || IsSMEAAccess(fault.accessdesc) then
        if HaltOnBreakpointOrWatchpoint() then
            if boolean IMPLEMENTATION_DEFINED "EDWAR is not valid on watchpoint debug event" then
                syndrome<10> = '1'; // FnV
            else
                if boolean IMPLEMENTATION_DEFINED "FAR is not valid on watchpoint exception" then
                    syndrome<10> = '1'; // FnV
        else
            if WatchpointFARNotPrecise(fault) then
                syndrome<15> = '1'; // FnP

    // Watchpoint number is valid if FEAT_Debugv8p9 is implemented or
    // if Feat_Debugv8p2 is implemented and below set of conditions are satisfied:
    // - Either FnV = 1 or FnP = 1.
    // - If the address recorded in FAR is not within a naturally-aligned block of memory.
    // Otherwise , it is IMPLEMENTATION DEFINED if watchpoint number is valid.
    if Havev8p9Debug() then
        syndrome<17> = '1'; // WPTV
        syndrome<23:18> = fault.watchpt_num<5:0>; // WPT
    elseif Havev82Debug() then
        if syndrome<15> == '1' || syndrome<10> == '1' then // Either of FnP or FnV is 1
            syndrome<17> = '1'; // WPTV
        elseif AddressNotInNaturallyAlignedBlock(vaddress) then // WPTV
            syndrome<17> = '1';
        elseif boolean IMPLEMENTATION_DEFINED "WPTV field is valid" then
            syndrome<17> = '1';
        if syndrome<17> == '1' then
            syndrome<23:18> = fault.watchpt_num<5:0>; // WPT
        else
            syndrome<23:18> = bits(6) UNKNOWN;

    return syndrome;
```

## Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictable

```
// ConstrainUnpredictable()
// =====
// Return the appropriate Constraint result to control the caller's behavior.
// The return value is IMPLEMENTATION DEFINED within a permitted list for each
// UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)

Constraint ConstrainUnpredictable(Unpredictable which);
```

### Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBits

```
// ConstrainUnpredictableBits()
// =====

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONSTRAINED.

(Constraint,bits(width)) ConstrainUnpredictableBits(Unpredictable which, integer width);
```

### Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBool

```
// ConstrainUnpredictableBool()
// =====
// This is a variant of the ConstrainUnpredictable function where the result is either
// Constraint_TRUE or Constraint_FALSE.

boolean ConstrainUnpredictableBool(Unpredictable which);
```

### Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableInteger

```
// ConstrainUnpredictableInteger()
// =====
// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns an UNKNOWN
// value in the range low to high, inclusive.

(Constraint,integer) ConstrainUnpredictableInteger(integer low, integer high,
Unpredictable which);
```

### Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableProcedure

```
// ConstrainUnpredictableProcedure()
// =====
// This is a variant of ConstrainUnpredictable that implements a Constrained
// Unpredictable behavior for a given Unpredictable situation.
// The behavior is within permitted behaviors for a given Unpredictable situation,
// these are documented in the textual part of the architecture specification.
//
// This function is expected to be refined in an IMPLEMENTATION DEFINED manner.
// The details of possible outcomes may not be present in the code and must be interpreted
// for each use with respect to the CONSTRAINED UNPREDICTABLE specifications
// for the specific area.

ConstrainUnpredictableProcedure(Unpredictable which);
```

## Library pseudocode for shared/functions/unpredictable/Constraint

```
// Constraint
// =====
// List of Constrained Unpredictable behaviors.

enumeration Constraint    { // General
    Constraint_NONE,      // Instruction executes with
                          // no change or side-effect
                          // to its described behavior
    Constraint_UNKNOWN,   // Destination register
                          // has UNKNOWN value
    Constraint_UNDEF,     // Instruction is UNDEFINED
    Constraint_UNDEFEL0,   // Instruction is UNDEFINED at EL0 only
    Constraint_NOP,        // Instruction executes as NOP
    Constraint_TRUE,
    Constraint_FALSE,
    Constraint_DISABLED,
    Constraint_UNCOND,     // Instruction executes unconditionally
    Constraint_COND,       // Instruction executes conditionally
    Constraint_ADDITIONAL_DECODE, // Instruction executes
                          // with additional decode

    // Load-store
    Constraint_WBSUPPRESS,
    Constraint_FAULT,
    Constraint_LIMITED_ATOMICITY, // Accesses are not
                          // single-copy atomic
                          // above the byte level

    Constraint_NVNV1_00,
    Constraint_NVNV1_01,
    Constraint_NVNV1_11,
    Constraint_EL1TIMESTAMP, // Constrain to Virtual Timestamp
    Constraint_EL2TIMESTAMP, // Constrain to Virtual Timestamp
    Constraint_OSH,          // Constrain to Outer Shareable
    Constraint_ISH,          // Constrain to Inner Shareable
    Constraint_NSH,          // Constrain to Nonshareable

    Constraint_NC,          // Constrain to Noncacheable
    Constraint_WT,          // Constrain to Writethrough
    Constraint_WB,          // Constrain to Writeback

    // IPA too large
    Constraint_FORCE, Constraint_FORCENOSLDCHECK,
    // An unallocated System register value maps onto an allocated value
    Constraint_MAPTOALLOCATED,
    // PMSCR_PCT reserved values select Virtual timestamp
    Constraint_PMSCR_PCT_VIRT
};
```





```

// Unpredictable
// =====
// List of Constrained Unpredictable situations.

enumeration Unpredictable {
    // VMSR on MVFR
    Unpredictable_VMSR,
    // Writeback/transfer register overlap (load)
    Unpredictable_WBOVERLAPLD,
    // Writeback/transfer register overlap (store)
    Unpredictable_WBOVERLAPST,
    // Load Pair transfer register overlap
    Unpredictable_LDPOVERLAP,
    // Store-exclusive base/status register overlap
    Unpredictable_BASEOVERLAP,
    // Store-exclusive data/status register overlap
    Unpredictable_DATAOVERLAP,
    // Load-store alignment checks
    Unpredictable_DEVPAGE2,
    // Instruction fetch from Device memory
    Unpredictable_INSTRDEVICE,
    // Reserved CPACR value
    Unpredictable_RESCPACR,
    // Reserved MAIR value
    Unpredictable_RESMAIR,
    // Effect of SCTLr_ELx.C on Tagged attribute
    Unpredictable_S1CTAGGED,
    // Reserved Stage 2 MemAttr value
    Unpredictable_S2RESMEMATTR,
    // Reserved TEX:C:B value
    Unpredictable_RESTEXCB,
    // Reserved PRRR value
    Unpredictable_RESRRR,
    // Reserved DACR field
    Unpredictable_RESDACR,
    // Reserved VTCR.S value
    Unpredictable_RESVTCRS,
    // Reserved TCR.TnSZ value
    Unpredictable_RESTnSZ,
    // Reserved SCTLr_ELx.TCF value
    Unpredictable_RESTCF,
    // Tag stored to Device memory
    Unpredictable_DEVICETAGSTORE,
    // Out-of-range TCR.TnSZ value
    Unpredictable_OORTnSZ,

    // IPA size exceeds PA size
    Unpredictable_LARGEIPA,
    // Syndrome for a known-passing conditional A32 instruction
    Unpredictable_ESRCONDPASS,
    // Illegal State exception: zero PSTATE.IT
    Unpredictable_ILZEROIT,
    // Illegal State exception: zero PSTATE.T
    Unpredictable_ILZEROT,
    // Debug: prioritization of Vector Catch
    Unpredictable_BPVECTORCATCHPRI,
    // Debug Vector Catch: match on 2nd halfword
    Unpredictable_VCMATCHHALF,
    // Debug Vector Catch: match on Data Abort
    // or Prefetch abort
    Unpredictable_VCMATCHDAPA,
    // Debug watchpoints: nonzero MASK and non-ones BAS
    Unpredictable_WPMASKANDBAS,
    // Debug watchpoints: non-contiguous BAS
    Unpredictable_WPBASCONTIGUOUS,
    // Debug watchpoints: reserved MASK
    Unpredictable_RESWPMASK,
    // Debug watchpoints: nonzero MASKed bits of address
    Unpredictable_WPMASKEDBITS,
    // Debug breakpoints and watchpoints: reserved control bits

```

```

Unpredictable_RESBPWPCTRL,
// Debug breakpoints: not implemented
Unpredictable_BPNOTIMPL,
// Debug breakpoints: reserved type
Unpredictable_RESBPTYPE,
// Debug breakpoints and watchpoints: reserved MDSELR_EL1.BANK
Unpredictable_RESMDSELR,
// Debug breakpoints: not-context-aware breakpoint
Unpredictable_BPNOTCTXCMP,
// Debug breakpoints: match on 2nd halfword of instruction
Unpredictable_BPMATCHHALF,
// Debug breakpoints: mismatch on 2nd halfword of instruction
Unpredictable_BPMISMATCHHALF,
// Debug breakpoints: a breakpoint is linked to that is not
// programmed with linking enabled
Unpredictable_BPLINKINGDISABLED,
// Debug breakpoints: reserved MASK
Unpredictable_RESBPMASK,
// Debug breakpoints: MASK is set for a Context matching
// breakpoint or when DBGBCR_EL1[n].BAS != '1111'
Unpredictable_BPMASK,
// Debug breakpoints: nonzero MASKed bits of address
Unpredictable_BPMASKEDBITS,
// Debug breakpoints: A linked breakpoint is
// linked to an address matching breakpoint
Unpredictable_BPLINKEDADDRMATCH,
// Debug: restart to a misaligned AArch32 PC value
Unpredictable_RESTARTALIGNPC,
// Debug: restart to a not-zero-extended AArch32 PC value
Unpredictable_RESTARTZEROUPPERPC,
// Zero top 32 bits of X registers in AArch32 state
Unpredictable_ZEROUPPER,
// Zero top 32 bits of PC on illegal return to
// AArch32 state
Unpredictable_ERETZEROUPPERPC,
// Force address to be aligned when interworking
// branch to A32 state
Unpredictable_A32FORCEALIGNPC,
// SMC disabled
Unpredictable_SMD,
// FF speculation
Unpredictable_NONFAULT,
// Zero top bits of Z registers in EL change
Unpredictable_SVEZEROUPPER,
// Load mem data in NF loads
Unpredictable_SVELDNFDATA,
// Write zeros in NF loads
Unpredictable_SVELDNFZERO,
// SP alignment fault when predicate is all zero
Unpredictable_CHECKSPNONEACTIVE,
// Zero top bits of ZA registers in EL change
Unpredictable_SMEZEROUPPER,
// Watchpoint match of last rounded up memory access in case of
// 16 byte rounding
Unpredictable_16BYTEROUNDEDUPACCESS,
// Watchpoint match of first rounded down memory access in case of
// 16 byte rounding
Unpredictable_16BYTEROUNDEDOWNACCESS,
// HCR_EL2.<NV,NV1> == '01'
Unpredictable_NVNV1,
// Reserved shareability encoding
Unpredictable_Shareability,
// Access Flag Update by HW
Unpredictable_AFUPDATE,
// Dirty Bit State Update by HW
Unpredictable_DBUPDATE,
// Consider SCTL_ELx[].IESB in Debug state
Unpredictable_IESBinDebug,
// Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
Unpredictable_BADPMSFCR,

```

```

// Zero saved BType value in SPSR_ELx/DPSR_EL0
Unpredictable_ZEROBTYPE,
// Timestamp constrained to virtual or physical
Unpredictable_EL2TIMESTAMP,
Unpredictable_EL1TIMESTAMP,
// Reserved MDCR_EL3.<NSTBE,NSTB> or MDCR_EL3.<NSPBE,NSPB> value
Unpredictable_RESERVEDNSxB,
// WFET or WFIT instruction in Debug state
Unpredictable_WFXTDEBUG,
// Address does not support LS64 instructions
Unpredictable_LS64UNSUPPORTED,
// Misaligned exclusives, atomics, acquire/release
// to region that is not Normal Cacheable WB
Unpredictable_MISALIGNEDATOMIC,
// 128-bit Atomic or 128-bit RCW{S} transfer register overlap
Unpredictable_LSE128OVERLAP,
// Clearing DCC/ITR sticky flags when instruction is in flight
Unpredictable_CLEARERRITEZERO,
// ALUEXCEPTIONRETURN when in user/system mode in
// A32 instructions
Unpredictable_ALUEXCEPTIONRETURN,
// Trap to register in debug state are ignored
Unpredictable_IGNORETRAPINDEBUG,
// Compare DBGBVR.RESS for BP/WP
Unpredictable_DBGxVR_RESS,
// Inaccessible event counter
Unpredictable_PMUEVENTCOUNTER,
// Reserved PMSCR.PCT behavior
Unpredictable_PMSCR_PCT,
// MDCR_EL2.HPMN or HDCR.HPMN is larger than PMCR.N or
// FEAT_HPMN0 is not implemented and HPMN is 0.
Unpredictable_CounterReservedForEL2,
// Generate BRB_FILTRATE event on BRB injection
Unpredictable_BRBFILTRATE,
// Generate PMU_SNAPSHOT event in Debug state
Unpredictable_PMUSNAPSHOTEVENT,
// Reserved MDCR_EL3.EPMSSAD value
Unpredictable_RESEPMSAD,
// Reserved PMECR_EL1.SSE value
Unpredictable_RESPMSSE,
// Enable for PMU exception and PMUIRQ
Unpredictable_RESPMEE,
// Operands for CPY*/SET* instructions overlap or
// use 0b11111 as a register specifier
Unpredictable_MOPSOVERLAP31,
// Store-only Tag checking on a failed Atomic Compare and Swap
Unpredictable_STOREONLYTAGCHECKEDCAS,
// Reserved MDCR_EL3.ETBAD value
Unpredictable_RES_ETBAD,
// accessing DBGDSRint via MRC in debug state
Unpredictable_MRC_APSR_TARGET,
// Reserved PMEVTYPER<n>_EL0.TC value
Unpredictable_RESTC

```

```
};
```

## Library pseudocode for shared/functions/vector/AdvSIMDExpandImm

```
// AdvSIMDExpandImm()
// =====

bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)
  bits(64) imm64;
  case cmode<3:1> of
    when '000'
      imm64 = Replicate(Zeros(24):imm8, 2);
    when '001'
      imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
    when '010'
      imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
    when '011'
      imm64 = Replicate(imm8:Zeros(24), 2);
    when '100'
      imm64 = Replicate(Zeros(8):imm8, 4);
    when '101'
      imm64 = Replicate(imm8:Zeros(8), 4);
    when '110'
      if cmode<0> == '0' then
        imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
      else
        imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
    when '111'
      if cmode<0> == '0' && op == '0' then
        imm64 = Replicate(imm8, 8);
      if cmode<0> == '0' && op == '1' then
        imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
        imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
        imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
        imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
        imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
      if cmode<0> == '1' && op == '0' then
        imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 5):imm8<5:0>:Zeros(19);
        imm64 = Replicate(imm32, 2);
      if cmode<0> == '1' && op == '1' then
        if UsingAArch32() then ReservedEncoding();
        imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 8):imm8<5:0>:Zeros(48);

  return imm64;
```

## Library pseudocode for shared/functions/vector/MatMulAdd

```
// MatMulAdd()
// =====
//
// Signed or unsigned 8-bit integer matrix multiply and add to 32-bit integer matrix
// result[2, 2] = addend[2, 2] + (op1[2, 8] * op2[8, 2])

bits(N) MatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, boolean op1_unsigned,
  boolean op2_unsigned)
  assert N == 128;

  bits(N) result;
  bits(32) sum;
  integer prod;

  for i = 0 to 1
    for j = 0 to 1
      sum = Elem[addend, 2*i + j, 32];
      for k = 0 to 7
        prod = (Int(Elem[op1, 8*i + k, 8], op1_unsigned) *
          Int(Elem[op2, 8*j + k, 8], op2_unsigned));
        sum = sum + prod;
      Elem[result, 2*i + j, 32] = sum;

  return result;
```

## Library pseudocode for shared/functions/vector/PolynomialMult

```
// PolynomialMult()
// =====

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = ZeroExtend(op2, M+N);
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;
```

## Library pseudocode for shared/functions/vector/SatQ

```
// SatQ()
// =====

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);
```

## Library pseudocode for shared/functions/vector/SignedSatQ

```
// SignedSatQ()
// =====

(bits(N), boolean) SignedSatQ(integer i, integer N)
    integer result;
    boolean saturated;
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1; saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1)); saturated = TRUE;
    else
        result = i; saturated = FALSE;
    return (result<N-1:0>, saturated);
```

## Library pseudocode for shared/functions/vector/UnsignedRSqrtEstimate

```
// UnsignedRSqrtEstimate()
// =====

bits(N) UnsignedRSqrtEstimate(bits(N) operand)
    assert N == 32;
    bits(N) result;
    if operand<N-1:N-2> == '00' then // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)
        // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
        increasedprecision = FALSE;
        estimate = RecipSqrtEstimate(UInt(operand<31:23>), increasedprecision);
        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

## Library pseudocode for shared/functions/vector/UnsignedRecipEstimate

```
// UnsignedRecipEstimate()
// =====

bits(N) UnsignedRecipEstimate(bits(N) operand)
    assert N == 32;
    bits(N) result;
    if operand<N-1> == '0' then // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)

        // estimate is in the range 256 to 511 representing [1.0 .. 2.0)
        increasedprecision = FALSE;
        estimate = RecipEstimate(UInt(operand<31:23>), increasedprecision);

        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

## Library pseudocode for shared/functions/vector/UnsignedSatQ

```
// UnsignedSatQ()
// =====

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    integer result;
    boolean saturated;
    if i > 2^N - 1 then
        result = 2^N - 1; saturated = TRUE;
    elsif i < 0 then
        result = 0; saturated = TRUE;
    else
        result = i; saturated = FALSE;
    return (result<N-1:0>, saturated);
```

## Library pseudocode for shared/trace/Common/DebugMemWrite

```
// DebugMemWrite()
// =====
// Write data to memory one byte at a time. Starting at the passed virtual address.
// Used by SPE.

(PhysMemRetStatus, AddressDescriptor) DebugMemWrite(bits(64) vaddress, AccessDescriptor accdesc,
                                                    boolean aligned, bits(8) data)

    PhysMemRetStatus memstatus = PhysMemRetStatus UNKNOWN;

    // Translate virtual address
    AddressDescriptor addrdesc;
    integer size = 1;
    addrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);

    if IsFault(addrdesc) then
        return (memstatus, addrdesc);

    memstatus = PhysMemWrite(addrdesc, 1, accdesc, data);

    return (memstatus, addrdesc);
```





```

// DebugWriteExternalAbort()
// =====
// Populate the syndrome register for an External abort caused by a call of DebugMemWrite().

DebugWriteExternalAbort(PhysMemRetStatus memstatus, AddressDescriptor addrdesc,
                        bits(64) start_vaddr)

    boolean iswrite = TRUE;

    boolean handle_as_SError = FALSE;
    boolean async_external_abort = FALSE;
    bits(64) syndrome;
    case addrdesc.fault.accessdesc.acctype of
        when AccessType\_SPE
            handle_as_SError = boolean IMPLEMENTATION_DEFINED "SPE SyncExternal as SError";
            async_external_abort = boolean IMPLEMENTATION_DEFINED "SPE async External abort";
            syndrome = PMBSR_EL1<63:0>;
        otherwise
            Unreachable();

    boolean ttw_abort;
    ttw_abort = addrdesc.fault.statuscode IN {Fault\_SyncExternalOnWalk,
                                              Fault\_SyncParityOnWalk};
    Fault statuscode = if ttw_abort then addrdesc.fault.statuscode else memstatus.statuscode;
    bit extflag = if ttw_abort then addrdesc.fault.extflag else memstatus.extflag;
    if (statuscode IN {Fault\_AsyncExternal, Fault\_AsyncParity} || handle_as_SError) then
        // ASYNC Fault -> SError or SYNC Fault handled as SError
        FaultRecord fault = NoFault();
        boolean parity = statuscode IN {Fault\_SyncParity, Fault\_AsyncParity,
                                       Fault\_SyncParityOnWalk};
        fault.statuscode = if parity then Fault\_AsyncParity else Fault\_AsyncExternal;
        if HaveRASExt() then
            fault.merrorstate = memstatus.merrorstate;
        fault.extflag = extflag;
        fault.accessdesc.acctype = addrdesc.fault.accessdesc.acctype;
        PendSErrorsInterrupt(fault);
    else
        // SYNC Fault, not handled by SError
        // Generate Buffer Management Event
        // EA bit
        syndrome<18> = '1';

        // DL bit for SPE
        if addrdesc.fault.accessdesc.acctype == AccessType\_SPE && (async_external_abort ||
            (start_vaddr != addrdesc.vaddress)) then
            syndrome<19> = '1';

        // Do not change following values if previous Buffer Management Event
        // has not been handled.
        // S bit
        if IsZero(syndrome<17>) then
            syndrome<17> = '1';

        // EC bits
        bits(6) ec;
        if (HaveRME() && addrdesc.fault.gpcf.gpf != GPCF\_None &&
            addrdesc.fault.gpcf.gpf != GPCF\_Fail) then
            ec = '011110';
        else
            ec = if addrdesc.fault.secondstage then '100101' else '100100';
        syndrome<31:26> = ec;

        // MSS bits
        if async_external_abort then
            syndrome<15:0> = Zeros(10) : '010001';
        else
            syndrome<15:0> = Zeros(10) : EncodeLDFSC(statuscode, addrdesc.fault.level);

    case addrdesc.fault.accessdesc.acctype of
        when AccessType\_SPE

```

```

        PMBSR_EL1<63:0> = syndrome;
    otherwise
        Unreachable();

```

## Library pseudocode for shared/trace/Common/DebugWriteFault

```

// DebugWriteFault()
// =====
// Populate the syndrome register for a Translation fault caused by a call of DebugMemWrite().

DebugWriteFault(bits(64) vaddress, FaultRecord fault)
    bits(64) syndrome;
    case fault.accessdesc.acctype of
        when AccessType\_SPE
            syndrome = PMBSR_EL1<63:0>;
        otherwise
            Unreachable();

    // MSS
    syndrome<15:0> = Zeros(10) : EncodeLDFSC(fault.statuscode, fault.level);

    // MSS2
    syndrome<55:32> = Zeros(24);

    // EC bits
    bits(6) ec;
    if HaveRME() && fault.gpcf.gpcf != GPCF\_None && fault.gpcf.gpcf != GPCF\_Fail then
        ec = '011110';
    else
        ec = if fault.secondstage then '100101' else '100100';
    syndrome<31:26> = ec;

    // S bit
    syndrome<17> = '1';

    if fault.statuscode == Fault\_Permission then
        // assuredonly bit
        syndrome<39> = if fault.assuredonly then '1' else '0';
        // overlay bit
        syndrome<38> = if fault.overlay then '1' else '0';
        // dirtybit
        syndrome<37> = if fault.dirtybit then '1' else '0';

    case fault.accessdesc.acctype of
        when AccessType\_SPE
            PMBSR_EL1<63:0> = syndrome;
        otherwise
            Unreachable();

    // Buffer Write Pointer already points to the address that generated the fault.
    // Writing to memory never started so no data loss. DL is unchanged.

    return;

```

## Library pseudocode for shared/trace/Common/GetTimestamp

```
// GetTimestamp()
// =====
// Returns the Timestamp depending on the type

bits(64) GetTimestamp(TimeStamp timeStampType)
    case timeStampType of
        when TimeStamp_Physical
            return PhysicalCountInt();
        when TimeStamp_Virtual
            return PhysicalCountInt() - CNTVOFF_EL2;
        when TimeStamp_OffsetPhysical
            bits(64) physoff = if PhysicalOffsetIsValid() then CNTPOFF_EL2 else Zeros(64);
            return PhysicalCountInt() - physoff;
        when TimeStamp_None
            return Zeros(64);
        when TimeStamp_CoreSight
            return bits(64) IMPLEMENTATION_DEFINED "CoreSight timestamp";
        otherwise
            Unreachable();
```

## Library pseudocode for shared/trace/Common/PhysicalOffsetIsValid

```
// PhysicalOffsetIsValid()
// =====
// Returns whether the Physical offset for the timestamp is valid

boolean PhysicalOffsetIsValid()
    if !HaveAArch64() then
        return FALSE;
    elsif !HaveEL(EL2) || !HaveECVExt() then
        return FALSE;
    elsif HaveEL(EL3) && SCR_EL3.NS == '1' && EffectiveSCR_EL3_RW() == '0' then
        return FALSE;
    elsif HaveEL(EL3) && SCR_EL3.ECVEN == '0' then
        return FALSE;
    elsif CNTHCTL_EL2.ECV == '0' then
        return FALSE;
    else
        return TRUE;
```

## Library pseudocode for shared/trace/TraceBranch/BranchNotTaken

```
// BranchNotTaken()
// =====
// Called when a branch is not taken.

BranchNotTaken(BranchType branchtype, boolean branch_conditional)
    boolean branchtaken = FALSE;
    if HaveStatisticalProfiling() then
        SPEBranch(bits(64) UNKNOWN, branchtype, branch_conditional, branchtaken);
    return;
```

## Library pseudocode for shared/trace/TraceBuffer/AllowExternalTraceBufferAccess

```
// AllowExternalTraceBufferAccess()
// =====
// Returns TRUE if an external debug interface access to the Trace Buffer
// registers is allowed, FALSE otherwise.
// The access may also be subject to OS Lock, power-down, etc.

boolean AllowExternalTraceBufferAccess()
    return AllowExternalTraceBufferAccess(AccessState());

// AllowExternalTraceBufferAccess()
// =====
// Returns TRUE if an external debug interface access to the Trace Buffer
// registers is allowed for the given Security state, FALSE otherwise.
// The access may also be subject to OS Lock, power-down, etc.

boolean AllowExternalTraceBufferAccess(SecurityState access_state)
    assert IsFeatureImplemented(FEAT_TRBE_EXT);
    assert IsFeatureImplemented(FEAT_Debugv8p4); // Required when Trace Buffer implemented

    bits(2) etbad = if HaveEL(EL3) then MDCR_EL3.ETBAD else '11';

    // Check for reserved values
    if !IsFeatureImplemented(FEAT_RME) && etbad IN {'01','10'} then
        Constraint c;
        (c, etbad) = ConstrainUnpredictableBits(Unpredictable\_RES\_ETBAD, 2);
        assert c IN {Constraint\_DISABLED, Constraint\_UNKNOWN};
        if c == Constraint\_DISABLED then return FALSE;
        // Otherwise the value returned by ConstrainUnpredictableBits must be a
        // not-reserved value

    case etbad of
        when '00'
            SecurityState ss = (if IsFeatureImplemented(FEAT_RME) then SS\_Root else SS\_Secure);
            return access_state == ss;
        when '01'
            return access_state IN {SS\_Root, SS\_Realm};
        when '10'
            return access_state IN {SS\_Root, SS\_Secure};
        when '11'
            return TRUE;
```

## Library pseudocode for shared/trace/TraceBuffer/TraceBufferEnabled

```
// TraceBufferEnabled()
// =====

boolean TraceBufferEnabled()
    if !HaveTraceBufferExtension() || TRBLIMITR_EL1.E == '0' then
        return FALSE;
    if !SelfHostedTraceEnabled() then
        return FALSE;
    (-, el) = TraceBufferOwner();
    return !ELUsingAArch32(el);
```

## Library pseudocode for shared/trace/TraceBuffer/TraceBufferOwner

```
// TraceBufferOwner()
// =====
// Return the owning Security state and Exception level. Must only be called
// when SelfHostedTraceEnabled() is TRUE.

(SecurityState, bits(2)) TraceBufferOwner()
    assert HaveTraceBufferExtension() && SelfHostedTraceEnabled();

    SecurityState owning_ss;
    if HaveEL(EL3) then
        bits(3) state_bits;
        if HaveRME() then
            state_bits = MDCR_EL3.<NSTBE,NSTB>;
            if (state_bits IN {'10x'} ||
                (!HaveSecureEL2Ext() && state_bits IN {'00x'})) then
                // Reserved value
                (-, state_bits) = ConstrainUnpredictableBits(Unpredictable_RESERVEDNSxB, 3);
        else
            state_bits = '0' : MDCR_EL3.NSTB;

        case state_bits of
            when '00x' owning_ss = SS_Secure;
            when '01x' owning_ss = SS_NonSecure;
            when '11x' owning_ss = SS_Realm;
    else
        owning_ss = if SecureOnlyImplementation() then SS_Secure else SS_NonSecure;
    bits(2) owning_el;
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        owning_el = if MDCR_EL2.E2TB == '00' then EL2 else EL1;
    else
        owning_el = EL1;
    return (owning_ss, owning_el);
```

## Library pseudocode for shared/trace/TraceBuffer/TraceBufferRunning

```
// TraceBufferRunning()
// =====

boolean TraceBufferRunning()
    return TraceBufferEnabled() && TRBSR_EL1.S == '0';
```

## Library pseudocode for shared/trace/TraceInstrumentationAllowed/TraceInstrumentationAllowed

```
// TraceInstrumentationAllowed()
// =====
// Returns TRUE if Instrumentation Trace is allowed
// in the given Exception level and Security state.

boolean TraceInstrumentationAllowed(SecurityState ss, bits(2) el)
    if !IsFeatureImplemented(FEAT_ITE) then return FALSE;
    if ELUsingAArch32(el) then return FALSE;

    if TraceAllowed(el) then
        bit ite_bit;
        case el of
            when EL3 ite_bit = '0';
            when EL2 ite_bit = TRCITECR_EL2.E2E;
            when EL1 ite_bit = TRCITECR_EL1.E1E;
            when EL0
                if EffectiveTGE() == '1' then
                    ite_bit = TRCITECR_EL2.E0HE;
                else
                    ite_bit = TRCITECR_EL1.E0E;

        if SelfHostedTraceEnabled() then
            return ite_bit == '1';
        else
            bit el_bit;
            bit ss_bit;
            case el of
                when EL0 el_bit = TRCITEEDCR.E0;
                when EL1 el_bit = TRCITEEDCR.E1;
                when EL2 el_bit = TRCITEEDCR.E2;
                when EL3 el_bit = TRCITEEDCR.E3;
            case ss of
                when SS Realm ss_bit = TRCITEEDCR.RL;
                when SS Secure ss_bit = TRCITEEDCR.S;
                when SS NonSecure ss_bit = TRCITEEDCR.NS;
                otherwise ss_bit = '1';

            boolean ed_allowed = ss_bit == '1' && el_bit == '1';

            if TRCCONFIGR.ITO == '1' then
                return ed_allowed;
            else
                return ed_allowed && ite_bit == '1';
        else
            return FALSE;
```

## Library pseudocode for shared/trace/selfhosted/EffectiveE0HTRE

```
// EffectiveE0HTRE()
// =====
// Returns effective E0HTRE value

bit EffectiveE0HTRE()
    return if ELUsingAArch32(EL2) then HTRFCR.E0HTRE else TRFCR_EL2.E0HTRE;
```

## Library pseudocode for shared/trace/selfhosted/EffectiveE0TRE

```
// EffectiveE0TRE()
// =====
// Returns effective E0TRE value

bit EffectiveE0TRE()
    return if ELUsingAArch32(EL1) then TRFCR.E0TRE else TRFCR_EL1.E0TRE;
```

## Library pseudocode for shared/trace/selfhosted/EffectiveE1TRE

```
// EffectiveE1TRE()
// =====
// Returns effective E1TRE value

bit EffectiveE1TRE()
    return if UsingAArch32() then TRFCR.E1TRE else TRFCR_EL1.E1TRE;
```

## Library pseudocode for shared/trace/selfhosted/EffectiveE2TRE

```
// EffectiveE2TRE()
// =====
// Returns effective E2TRE value

bit EffectiveE2TRE()
    return if UsingAArch32() then HTRFCR.E2TRE else TRFCR_EL2.E2TRE;
```

## Library pseudocode for shared/trace/selfhosted/SelfHostedTraceEnabled

```
// SelfHostedTraceEnabled()
// =====
// Returns TRUE if Self-hosted Trace is enabled.

boolean SelfHostedTraceEnabled()
    bit secure_trace_enable = '0';
    if !(HaveTraceExt() && HaveSelfHostedTrace()) then return FALSE;
    if EDSCR.TFO == '0' then return TRUE;
    if HaveRME() then
        secure_trace_enable = if HaveSecureEL2Ext() then MDCR_EL3.STE else '0';
        return ((secure_trace_enable == '1' && !ExternalSecureNoninvasiveDebugEnabled()) ||
            (MDCR_EL3.RLTE == '1' && !ExternalRealmNoninvasiveDebugEnabled()));
    if HaveEL(EL3) then
        secure_trace_enable = if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE;
    else
        secure_trace_enable = if SecureOnlyImplementation() then '1' else '0';

    if secure_trace_enable == '1' && !ExternalSecureNoninvasiveDebugEnabled() then
        return TRUE;

    return FALSE;
```

## Library pseudocode for shared/trace/selfhosted/TraceAllowed

```
// TraceAllowed()
// =====
// Returns TRUE if Self-hosted Trace is allowed in the given Exception level.

boolean TraceAllowed(bits(2) el)
    if !HaveTraceExt() then return FALSE;
    if SelfHostedTraceEnabled() then
        boolean trace_allowed;
        ss = SecurityStateAtEL(el);
        // Detect scenarios where tracing in this Security state is never allowed.
        case ss of
            when SS_NonSecure
                trace_allowed = TRUE;
            when SS_Secure
                bit trace_bit;
                if HaveEL(EL3) then
                    trace_bit = if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE;
                else
                    trace_bit = '1';
                trace_allowed = trace_bit == '1';
            when SS_Realm
                trace_allowed = MDCR_EL3.RLTE == '1';
            when SS_Root
                trace_allowed = FALSE;

        // Tracing is prohibited if the trace buffer owning security state is not the
        // current Security state or the owning Exception level is a lower Exception level.
        if HaveTraceBufferExtension() && TraceBufferEnabled() then
            (owning_ss, owning_el) = TraceBufferOwner();
            if (ss != owning_ss || UInt(owning_el) < UInt(el) ||
                (EffectiveTGE() == '1' && owning_el == EL1)) then
                trace_allowed = FALSE;

    bit TRE_bit;
    case el of
        when EL3 TRE_bit = if !HaveAArch64() then TRFCR.E1TRE else '0';
        when EL2 TRE_bit = EffectiveE2TRE();
        when EL1 TRE_bit = EffectiveE1TRE();
        when EL0
            if EffectiveTGE() == '1' then
                TRE_bit = EffectiveE0HTRE();
            else
                TRE_bit = EffectiveE0TRE();

    return trace_allowed && TRE_bit == '1';
else
    return ExternalNoninvasiveDebugAllowed(el);
```

## Library pseudocode for shared/trace/selfhosted/TraceContextIDR2

```
// TraceContextIDR2()
// =====

boolean TraceContextIDR2()
    if !TraceAllowed(PSTATE.EL) || !HaveEL(EL2) then return FALSE;
    return (!SelfHostedTraceEnabled() || TRFCR_EL2.CX == '1');
```

## Library pseudocode for shared/trace/selfhosted/TraceSynchronizationBarrier

```
// TraceSynchronizationBarrier()
// =====
// Memory barrier instruction that preserves the relative order of memory accesses to System
// registers due to trace operations and other memory accesses to the same registers

TraceSynchronizationBarrier();
```



## Library pseudocode for shared/trace/selfhosted/TraceTimeStamp

```
// TraceTimeStamp()
// =====

TimeStamp TraceTimeStamp()
    if SelfHostedTraceEnabled() then
        if HaveEL(EL2) then
            TS_el2 = TRFCR_EL2.TS;
            if !HaveECVExt() && TS_el2 == '10' then
                // Reserved value
                (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable\_EL2TIMESTAMP, 2);

            case TS_el2 of
                when '00'
                    // Falls out to check TRFCR_EL1.TS
                when '01'
                    return TimeStamp\_Virtual;
                when '10'
                    assert HaveECVExt(); // Otherwise ConstrainUnpredictableBits removes this case
                    return TimeStamp\_OffsetPhysical;
                when '11'
                    return TimeStamp\_Physical;

            TS_el1 = TRFCR_EL1.TS;
            if TS_el1 == '00' || (!HaveECVExt() && TS_el1 == '10') then
                // Reserved value
                (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable\_EL1TIMESTAMP, 2);

            case TS_el1 of
                when '01'
                    return TimeStamp\_Virtual;
                when '10'
                    assert HaveECVExt();
                    return TimeStamp\_OffsetPhysical;
                when '11'
                    return TimeStamp\_Physical;
                otherwise
                    Unreachable(); // ConstrainUnpredictableBits removes this case
            else
                return TimeStamp\_CoreSight;
```

## Library pseudocode for shared/trace/system/IsTraceCorePowered

```
// IsTraceCorePowered()
// =====
// Returns TRUE if the Trace Core Power Domain is powered up

boolean IsTraceCorePowered();
```

## Library pseudocode for shared/translation/at

```
enumeration TranslationStage {
    TranslationStage_1,
    TranslationStage_12
};

enumeration ATAccess {
    ATAccess_Read,
    ATAccess_Write,
    ATAccess_Any,
    ATAccess_ReadPAN,
    ATAccess_WritePAN
};
```

## Library pseudocode for shared/translation/at/EncodePARAttr

```
// EncodePARAttr()
// =====
// Convert orthogonal attributes and hints to 64-bit PAR ATTR field.

bits(8) EncodePARAttr(MemoryAttributes memattr)
    bits(8) result;

    if HaveMTEExt() && memattr.tags == MemTag_AllocationTagged then
        if HaveMTEPermExt() && memattr.notagaccess then
            result<7:0> = '11100000';
        else
            result<7:0> = '11110000';
        return result;

    if memattr.memtype == MemType_Device then
        result<7:4> = '0000';
        case memattr.device of
            when DeviceType_nGnRnE result<3:0> = '0000';
            when DeviceType_nGnRE result<3:0> = '0100';
            when DeviceType_nGRE result<3:0> = '1000';
            when DeviceType_GRE result<3:0> = '1100';
            otherwise Unreachable();
        result<0> = NOT memattr.xs;
    else
        if memattr.xs == '0' then
            if (memattr.outer.attrs == MemAttr_WT && memattr.inner.attrs == MemAttr_WT &&
                !memattr.outer.transient && memattr.outer.hints == MemHint_RA) then
                return '10100000';
            elsif memattr.outer.attrs == MemAttr_NC && memattr.inner.attrs == MemAttr_NC then
                return '01000000';

            if memattr.outer.attrs == MemAttr_WT then
                result<7:6> = if memattr.outer.transient then '00' else '10';
                result<5:4> = memattr.outer.hints;
            elsif memattr.outer.attrs == MemAttr_WB then
                result<7:6> = if memattr.outer.transient then '01' else '11';
                result<5:4> = memattr.outer.hints;
            else // MemAttr_NC
                result<7:4> = '0100';

            if memattr.inner.attrs == MemAttr_WT then
                result<3:2> = if memattr.inner.transient then '00' else '10';
                result<1:0> = memattr.inner.hints;
            elsif memattr.inner.attrs == MemAttr_WB then
                result<3:2> = if memattr.inner.transient then '01' else '11';
                result<1:0> = memattr.inner.hints;
            else // MemAttr_NC
                result<3:0> = '0100';

        return result;
```

## Library pseudocode for shared/translation/at/PAREncodeShareability

```
// PAREncodeShareability()
// =====
// Derive 64-bit PAR SH field.

bits(2) PAREncodeShareability(MemoryAttributes memattrs)
    if (memattrs.memtype == MemType\_Device ||
        (memattrs.inner.attrs == MemAttr\_NC &&
         memattrs.outer.attrs == MemAttr\_NC)) then
        // Force Outer-Shareable on Device and Normal Non-Cacheable memory
        return '10';

    case memattrs.shareability of
        when Shareability\_NSH return '00';
        when Shareability\_ISH return '11';
        when Shareability\_OSH return '10';
```

## Library pseudocode for shared/translation/at/ReportedPARAttrs

```
// ReportedPARAttrs()
// =====
// The value returned in this field can be the resulting attribute, as determined by any permitted
// implementation choices and any applicable configuration bits, instead of the value that appears
// in the translation table descriptor.

bits(8) ReportedPARAttrs(bits(8) parattrs);
```

## Library pseudocode for shared/translation/at/ReportedPARShareability

```
// ReportedPARShareability()
// =====
// The value returned in SH field can be the resulting attribute, as determined by any
// permitted implementation choices and any applicable configuration bits, instead of
// the value that appears in the translation table descriptor.

bits(2) ReportedPARShareability(bits(2) sh);
```

## Library pseudocode for shared/translation/attrs/DecodeDevice

```
// DecodeDevice()
// =====
// Decode output Device type

DeviceType DecodeDevice(bits(2) device)
    case device of
        when '00' return DeviceType\_nGnRnE;
        when '01' return DeviceType\_nGnRE;
        when '10' return DeviceType\_nGRE;
        when '11' return DeviceType\_GRE;
```

## Library pseudocode for shared/translation/attrs/DecodeLDFAttr

```
// DecodeLDFAttr()
// =====
// Decode memory attributes using LDF (Long Descriptor Format) mapping

MemAttrHints DecodeLDFAttr(bits(4) attr)
    MemAttrHints ldfattr;

    if attr IN {'x0xx'} then ldfattr.attrs = MemAttr\_WT; // Write-through
    elsif attr == '0100' then ldfattr.attrs = MemAttr\_NC; // Non-cacheable
    elsif attr IN {'x1xx'} then ldfattr.attrs = MemAttr\_WB; // Write-back
    else
        Unreachable();

    // Allocation hints are applicable only to cacheable memory.
    if ldfattr.attrs != MemAttr\_NC then
        case attr<1:0> of
            when '00' ldfattr.hints = MemHint\_No; // No allocation hints
            when '01' ldfattr.hints = MemHint\_WA; // Write-allocate
            when '10' ldfattr.hints = MemHint\_RA; // Read-allocate
            when '11' ldfattr.hints = MemHint\_RWA; // Read/Write allocate

    // The Transient hint applies only to cacheable memory with some allocation hints.
    if ldfattr.attrs != MemAttr\_NC && ldfattr.hints != MemHint\_No then
        ldfattr.transient = attr<3> == '0';

    return ldfattr;
```

## Library pseudocode for shared/translation/attrs/DecodeSDFAttr

```
// DecodeSDFAttr()
// =====
// Decode memory attributes using SDF (Short Descriptor Format) mapping

MemAttrHints DecodeSDFAttr(bits(2) rgn)
    MemAttrHints sdfattr;

    case rgn of
        when '00' // Non-cacheable (no allocate)
            sdfattr.attrs = MemAttr\_NC;
        when '01' // Write-back, Read and Write allocate
            sdfattr.attrs = MemAttr\_WB;
            sdfattr.hints = MemHint\_RWA;
        when '10' // Write-through, Read allocate
            sdfattr.attrs = MemAttr\_WT;
            sdfattr.hints = MemHint\_RA;
        when '11' // Write-back, Read allocate
            sdfattr.attrs = MemAttr\_WB;
            sdfattr.hints = MemHint\_RA;

    sdfattr.transient = FALSE;

    return sdfattr;
```

## Library pseudocode for shared/translation/attrs/DecodeShareability

```
// DecodeShareability()
// =====
// Decode shareability of target memory region

Shareability DecodeShareability(bits(2) sh)
    case sh of
        when '10' return Shareability\_OSH;
        when '11' return Shareability\_ISH;
        when '00' return Shareability\_NSH;
        otherwise
            case ConstrainUnpredictable(Unpredictable Shareability) of
                when Constraint\_OSH return Shareability\_OSH;
                when Constraint\_ISH return Shareability\_ISH;
                when Constraint\_NSH return Shareability\_NSH;
```

## Library pseudocode for shared/translation/attrs/EffectiveShareability

```
// EffectiveShareability()
// =====
// Force Outer Shareability on Device and Normal iNCoNC memory

Shareability EffectiveShareability(MemoryAttributes memattrs)
    if (memattrs.memtype == MemType\_Device ||
        (memattrs.inner.attrs == MemAttr\_NC &&
         memattrs.outer.attrs == MemAttr\_NC)) then
        return Shareability\_OSH;
    else
        return memattrs.shareability;
```

## Library pseudocode for shared/translation/attrs/NormalNCMemAttr

```
// NormalNCMemAttr()
// =====
// Normal Non-cacheable memory attributes

MemoryAttributes NormalNCMemAttr()
    MemAttrHints non_cacheable;
    non_cacheable.attrs = MemAttr\_NC;

    MemoryAttributes nc_memattrs;
    nc_memattrs.memtype      = MemType\_Normal;
    nc_memattrs.outer        = non_cacheable;
    nc_memattrs.inner        = non_cacheable;
    nc_memattrs.shareability = Shareability\_OSH;
    nc_memattrs.tags         = MemTag\_Untagged;
    nc_memattrs.notagaccess  = FALSE;

    return nc_memattrs;
```

## Library pseudocode for shared/translation/attrs/S1ConstrainUnpredictableRESMAIR

```
// S1ConstrainUnpredictableRESMAIR()
// =====
// Determine whether a reserved value occupies MAIR_ELx.AttrN

boolean S1ConstrainUnpredictableRESMAIR(bits(8) attr, boolean slaarch64)
    case attr of
        when '0000xx01' return !(slaarch64 && HaveFeatXS());
        when '0000xxxx' return attr<1:0> != '00';
        when '01000000' return !(slaarch64 && HaveFeatXS());
        when '10100000' return !(slaarch64 && HaveFeatXS());
        when '11110000' return !(slaarch64 && HaveMTE2Ext());
        when 'xxxx0000' return TRUE;
        otherwise       return FALSE;
```

## Library pseudocode for shared/translation/attrs/S1DecodeMemAttrs

```
// S1DecodeMemAttrs()
// =====
// Decode MAIR-format memory attributes assigned in stage 1

MemoryAttributes S1DecodeMemAttrs(bits(8) attr_in, bits(2) sh, boolean slaarch64,
                                   S1TTWParams walkparams)

bits(8) attr = attr_in;
if S1ConstrainUnpredictableRESMAIR(attr, slaarch64) then
    (-, attr) = ConstrainUnpredictableBits(Unpredictable\_RESMAIR, 8);

MemoryAttributes memattrs;
case attr of
    when '0000xxxx' // Device memory
        memattrs.memtype = MemType\_Device;
        memattrs.device  = DecodeDevice(attr<3:2>);
        memattrs.xs      = if slaarch64 then NOT attr<0> else '1';
    when '01000000'
        assert slaarch64 && HaveFeatXS();
        memattrs.memtype = MemType\_Normal;
        memattrs.outer.attrs = MemAttr\_NC;
        memattrs.inner.attrs = MemAttr\_NC;
        memattrs.xs          = '0';

    when '10100000'
        assert slaarch64 && HaveFeatXS();
        memattrs.memtype = MemType\_Normal;
        memattrs.outer.attrs = MemAttr\_WT;
        memattrs.outer.hints = MemHint\_RA;
        memattrs.outer.transient = FALSE;
        memattrs.inner.attrs = MemAttr\_WT;
        memattrs.inner.hints = MemHint\_RA;
        memattrs.inner.transient = FALSE;
        memattrs.xs          = '0';
    when '11110000' // Tagged memory
        assert slaarch64 && HaveMTE2Ext();
        memattrs.memtype = MemType\_Normal;
        memattrs.outer.attrs = MemAttr\_WB;
        memattrs.outer.hints = MemHint\_RWA;
        memattrs.outer.transient = FALSE;
        memattrs.inner.attrs = MemAttr\_WB;
        memattrs.inner.hints = MemHint\_RWA;
        memattrs.inner.transient = FALSE;
        memattrs.xs          = '0';
    otherwise
        memattrs.memtype = MemType\_Normal;
        memattrs.outer  = DecodeLDFAttr(attr<7:4>);
        memattrs.inner  = DecodeLDFAttr(attr<3:0>);

        if (memattrs.inner.attrs == MemAttr\_WB &&
            memattrs.outer.attrs == MemAttr\_WB) then
            memattrs.xs = '0';
        else
            memattrs.xs = '1';

    if slaarch64 && attr IN {'11110000'} then
        memattrs.tags = MemTag\_AllocationTagged;
    elsif slaarch64 && walkparams.mtx == '1' then
        memattrs.tags = MemTag\_CanonicallyTagged;
    else
        memattrs.tags = MemTag\_Untagged;

    memattrs.notagaccess = FALSE;

    memattrs.shareability = DecodeShareability(sh);

return memattrs;
```

## Library pseudocode for shared/translation/attrs/S2CombineS1AttrHints

```
// S2CombineS1AttrHints()
// =====
// Determine resultant Normal memory cacheability and allocation hints from
// combining stage 1 Normal memory attributes and stage 2 cacheability attributes.

MemAttrHints S2CombineS1AttrHints(MemAttrHints s1_attrhints, MemAttrHints s2_attrhints)
    MemAttrHints attrhints;

    if s1_attrhints.attrs == MemAttr_NC || s2_attrhints.attrs == MemAttr_NC then
        attrhints.attrs = MemAttr_NC;
    elsif s1_attrhints.attrs == MemAttr_WT || s2_attrhints.attrs == MemAttr_WT then
        attrhints.attrs = MemAttr_WT;
    else
        attrhints.attrs = MemAttr_WB;

    // Stage 2 does not assign any allocation hints
    // Instead, they are inherited from stage 1
    if attrhints.attrs != MemAttr_NC then
        attrhints.hints      = s1_attrhints.hints;
        attrhints.transient = s1_attrhints.transient;

    return attrhints;
```

## Library pseudocode for shared/translation/attrs/S2CombineS1Device

```
// S2CombineS1Device()
// =====
// Determine resultant Device type from combining output memory attributes
// in stage 1 and Device attributes in stage 2

DeviceType S2CombineS1Device(DeviceType s1_device, DeviceType s2_device)
    if s1_device == DeviceType_nGnRnE || s2_device == DeviceType_nGnRnE then
        return DeviceType_nGnRnE;
    elsif s1_device == DeviceType_nGnRE || s2_device == DeviceType_nGnRE then
        return DeviceType_nGnRE;
    elsif s1_device == DeviceType_nGRE || s2_device == DeviceType_nGRE then
        return DeviceType_nGRE;
    else
        return DeviceType_GRE;
```

## Library pseudocode for shared/translation/attrs/S2CombineS1MemAttrs

```
// S2CombineS1MemAttrs()
// =====
// Combine stage 2 with stage 1 memory attributes

MemoryAttributes S2CombineS1MemAttrs(MemoryAttributes s1_memattrs, MemoryAttributes s2_memattrs,
                                     boolean s2aarch64)

    MemoryAttributes memattrs;

    if s1_memattrs.memtype == MemType_Device && s2_memattrs.memtype == MemType_Device then
        memattrs.memtype = MemType_Device;
        memattrs.device = S2CombineS1Device(s1_memattrs.device, s2_memattrs.device);
    elseif s1_memattrs.memtype == MemType_Device then // S2 Normal, S1 Device
        memattrs = s1_memattrs;
    elseif s2_memattrs.memtype == MemType_Device then // S2 Device, S1 Normal
        memattrs = s2_memattrs;
    else // S2 Normal, S1 Normal
        memattrs.memtype = MemType_Normal;
        memattrs.inner = S2CombineS1AttrHints(s1_memattrs.inner, s2_memattrs.inner);
        memattrs.outer = S2CombineS1AttrHints(s1_memattrs.outer, s2_memattrs.outer);

    memattrs.tags = S2MemTagType(memattrs, s1_memattrs.tags);

    if !HaveMTEPermExt() then
        memattrs.notagaccess = FALSE;
    else
        memattrs.notagaccess = (s2_memattrs.notagaccess &&
                                s1_memattrs.tags == MemTag_AllocationTagged);
    memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability,
                                                    s2_memattrs.shareability);

    if (memattrs.memtype == MemType_Normal &&
        memattrs.inner.attrs == MemAttr_WB &&
        memattrs.outer.attrs == MemAttr_WB) then
        memattrs.xs = '0';
    elseif s2aarch64 then
        memattrs.xs = s2_memattrs.xs AND s1_memattrs.xs;
    else
        memattrs.xs = s1_memattrs.xs;

    memattrs.shareability = EffectiveShareability(memattrs);
    return memattrs;
```

## Library pseudocode for shared/translation/attrs/S2CombineS1Shareability

```
// S2CombineS1Shareability()
// =====
// Combine stage 2 shareability with stage 1

Shareability S2CombineS1Shareability(Shareability s1_shareability,
                                     Shareability s2_shareability)

    if (s1_shareability == Shareability_OSH ||
        s2_shareability == Shareability_OSH) then
        return Shareability_OSH;
    elseif (s1_shareability == Shareability_ISH ||
            s2_shareability == Shareability_ISH) then
        return Shareability_ISH;
    else
        return Shareability_NSH;
```



## Library pseudocode for shared/translation/attrs/S2DecodeCacheability

```
// S2DecodeCacheability()
// =====
// Determine the stage 2 cacheability for Normal memory

MemAttrHints S2DecodeCacheability(bits(2) attr)
    MemAttrHints s2attr;

    case attr of
        when '01' s2attr.attrs = MemAttr_NC; // Non-cacheable
        when '10' s2attr.attrs = MemAttr_WT; // Write-through
        when '11' s2attr.attrs = MemAttr_WB; // Write-back
        otherwise // Constrained unpredictable
            case ConstrainUnpredictable(Unpredictable S2RESMEMATTR) of
                when Constraint_NC s2attr.attrs = MemAttr_NC;
                when Constraint_WT s2attr.attrs = MemAttr_WT;
                when Constraint_WB s2attr.attrs = MemAttr_WB;

    // Stage 2 does not assign hints or the transient property
    // They are inherited from stage 1 if the result of the combination allows it
    s2attr.hints = bits(2) UNKNOWN;
    s2attr.transient = boolean UNKNOWN;

    return s2attr;
```

## Library pseudocode for shared/translation/attrs/S2DecodeMemAttrs

```
// S2DecodeMemAttrs()
// =====
// Decode stage 2 memory attributes

MemoryAttributes S2DecodeMemAttrs(bits(4) attr, bits(2) sh, boolean s2aarch64)
    MemoryAttributes memattrs;

    case attr of
        when '00xx' // Device memory
            memattrs.memtype = MemType_Device;
            memattrs.device = DecodeDevice(attr<1:0>);
        when '0100' // Normal, Inner+Outer WB cacheable NoTagAccess memory
            if s2aarch64 && HaveMTEPermExt() then
                memattrs.memtype = MemType_Normal;
                memattrs.outer = S2DecodeCacheability('11'); // Write-back
                memattrs.inner = S2DecodeCacheability('11'); // Write-back
            else
                memattrs.memtype = MemType_Normal;
                memattrs.outer = S2DecodeCacheability(attr<3:2>);
                memattrs.inner = S2DecodeCacheability(attr<1:0>);
        otherwise // Normal memory
            memattrs.memtype = MemType_Normal;
            memattrs.outer = S2DecodeCacheability(attr<3:2>);
            memattrs.inner = S2DecodeCacheability(attr<1:0>);

    memattrs.shareability = DecodeShareability(sh);

    if s2aarch64 && HaveMTEPermExt() then
        memattrs.notagaccess = attr == '0100';
    else
        memattrs.notagaccess = FALSE;

    return memattrs;
```

## Library pseudocode for shared/translation/attrs/S2MemTagType

```
// S2MemTagType()
// =====
// Determine whether the combined output memory attributes of stage 1 and
// stage 2 indicate tagged memory

MemTagType S2MemTagType(MemoryAttributes s2_memattrs, MemTagType s1_tagtype)

    if !HaveMTE2Ext() then
        return MemTag_Untagged;

    if ((s1_tagtype == MemTag_AllocationTagged) &&
        (s2_memattrs.memtype == MemType_Normal) &&
        (s2_memattrs.inner.attrs == MemAttr_WB) &&
        (s2_memattrs.inner.hints == MemHint_RWA) &&
        (!s2_memattrs.inner.transient) &&
        (s2_memattrs.outer.attrs == MemAttr_WB) &&
        (s2_memattrs.outer.hints == MemHint_RWA) &&
        (!s2_memattrs.outer.transient)) then
        return MemTag_AllocationTagged;

    // Return what stage 1 asked for if we can, otherwise Untagged.
    if s1_tagtype != MemTag_AllocationTagged then
        return s1_tagtype;

    return MemTag_Untagged;
```

## Library pseudocode for shared/translation/attrs/WalkMemAttrs

```
// WalkMemAttrs()
// =====
// Retrieve memory attributes of translation table walk

MemoryAttributes WalkMemAttrs(bits(2) sh, bits(2) irgn, bits(2) orgn)
    MemoryAttributes walkmemattrs;

    walkmemattrs.memtype = MemType_Normal;
    walkmemattrs.shareability = DecodeShareability(sh);
    walkmemattrs.inner = DecodeSDFAttr(irgn);
    walkmemattrs.outer = DecodeSDFAttr(orn);
    walkmemattrs.tags = MemTag_Untagged;
    if (walkmemattrs.inner.attrs == MemAttr_WB &&
        walkmemattrs.outer.attrs == MemAttr_WB) then
        walkmemattrs.xs = '0';
    else
        walkmemattrs.xs = '1';
    walkmemattrs.notagaccess = FALSE;

    return walkmemattrs;
```

## Library pseudocode for shared/translation/faults/AlignmentFault

```
// AlignmentFault()
// =====
// Return a fault record indicating an Alignment fault not due to memory type has occurred
// for a specific access

FaultRecord AlignmentFault(AccessDescriptor accdesc)
    FaultRecord fault;

    fault.statuscode = Fault\_Alignment;
    fault.accessdesc = accdesc;
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;
    fault.write = !accdesc.read && accdesc.write;
    fault.gpcfs2walk = FALSE;
    fault.gpcf = GPCNoFault();

    return fault;
```

## Library pseudocode for shared/translation/faults/ExclusiveFault

```
// ExclusiveFault()
// =====
// Return a fault record indicating an Exclusive fault for a specific access

FaultRecord ExclusiveFault(AccessDescriptor accdesc)
    FaultRecord fault;

    fault.statuscode = Fault\_Exclusive;
    fault.accessdesc = accdesc;
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;
    fault.write = !accdesc.read && accdesc.write;
    fault.gpcfs2walk = FALSE;
    fault.gpcf = GPCNoFault();

    return fault;
```

## Library pseudocode for shared/translation/faults/NoFault

```
// NoFault()
// =====
// Return a clear fault record indicating no faults have occurred

FaultRecord NoFault()
    FaultRecord fault;

    fault.statuscode = Fault\_None;
    fault.accessdesc = AccessDescriptor UNKNOWN;
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;
    fault.dirtybit = FALSE;
    fault.overlay = FALSE;
    fault.toplevel = FALSE;
    fault.assuredonly = FALSE;
    fault.s1tagnotdata = FALSE;
    fault.tagaccess = FALSE;
    fault.gpcfs2walk = FALSE;
    fault.gpcf = GPCNoFault();

    return fault;

// NoFault()
// =====
// Return a clear fault record indicating no faults have occurred for a specific access

FaultRecord NoFault(AccessDescriptor accdesc)
    FaultRecord fault;

    fault.statuscode = Fault\_None;
    fault.accessdesc = accdesc;
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;
    fault.dirtybit = FALSE;
    fault.overlay = FALSE;
    fault.toplevel = FALSE;
    fault.assuredonly = FALSE;
    fault.s1tagnotdata = FALSE;
    fault.tagaccess = FALSE;
    fault.write = !accdesc.read && accdesc.write;
    fault.gpcfs2walk = FALSE;
    fault.gpcf = GPCNoFault();

    return fault;
```

## Library pseudocode for shared/translation/gpc/AbovePPS

```
// AbovePPS()
// =====
// Returns TRUE if an address exceeds the range configured in GPCCR_EL3.PPS.

boolean AbovePPS(bits(56) address)
    pps = DecodePPS();
    if pps >= 56 then
        return FALSE;

    return IsZero(address<55:pps>);
```

## Library pseudocode for shared/translation/gpc/DecodeGPTBlock

```
// DecodeGPTBlock()
// =====
// Validate and decode a GPT Block descriptor

(GPCF, GPTEntry) DecodeGPTBlock(PGSe pgs, bits(64) gpt_entry)
    assert gpt_entry<3:0> == GPT\_Block;
    GPTEntry result;

    if !IsZero(gpt_entry<63:8>) then
        return (GPCF\_Walk, GPTEntry UNKNOWN);

    if !GPIValid(gpt_entry<7:4>) then
        return (GPCF\_Walk, GPTEntry UNKNOWN);

    result.gpi = gpt_entry<7:4>;
    result.level = 0;

    // GPT information from a level 0 GPT Block descriptor is permitted
    // to be cached in a TLB as though the Block is a contiguous region
    // of granules each of the size configured in GPCCR_EL3.PGS.
    case pgs of
        when PGS\_4KB result.size = GPTRange\_4KB;
        when PGS\_16KB result.size = GPTRange\_16KB;
        when PGS\_64KB result.size = GPTRange\_64KB;
        otherwise Unreachable();
    result.contig_size = GPTL0Size();

    return (GPCF\_None, result);
```

## Library pseudocode for shared/translation/gpc/DecodeGPTContiguous

```
// DecodeGPTContiguous()
// =====
// Validate and decode a GPT Contiguous descriptor

(GPCF, GPTEntry) DecodeGPTContiguous(PGSe pgs, bits(64) gpt_entry)
    assert gpt_entry<3:0> == GPT\_Contig;
    GPTEntry result;

    if !IsZero(gpt_entry<63:10>) then
        return (GPCF\_Walk, result);

    result.gpi = gpt_entry<7:4>;
    if !GPIValid(result.gpi) then
        return (GPCF\_Walk, result);

    case pgs of
        when PGS\_4KB result.size = GPTRange\_4KB;
        when PGS\_16KB result.size = GPTRange\_16KB;
        when PGS\_64KB result.size = GPTRange\_64KB;
        otherwise Unreachable();

    case gpt_entry<9:8> of
        when '01' result.contig_size = GPTRange\_2MB;
        when '10' result.contig_size = GPTRange\_32MB;
        when '11' result.contig_size = GPTRange\_512MB;
        otherwise return (GPCF\_Walk, GPTEntry UNKNOWN);

    result.level = 1;

    return (GPCF\_None, result);
```

## Library pseudocode for shared/translation/gpc/DecodeGPTGranules

```
// DecodeGPTGranules()
// =====
// Validate and decode a GPT Granules descriptor

(GPCF, GPTEntry) DecodeGPTGranules(PGSe pgs, integer index, bits(64) gpt_entry)
GPTEntry result;

for i = 0 to 15
    if !GPIValid(gpt_entry<i*4 +:4>) then
        return (GPCF\_Walk, result);

result.gpi = gpt_entry<index*4 +:4>;

case pgs of
    when PGS\_4KB result.size = GPTRange\_4KB;
    when PGS\_16KB result.size = GPTRange\_16KB;
    when PGS\_64KB result.size = GPTRange\_64KB;
    otherwise Unreachable();

result.contig_size = result.size; // No contiguity
result.level = 1;

return (GPCF\_None, result);
```

## Library pseudocode for shared/translation/gpc/DecodeGPTTable

```
// DecodeGPTTable()
// =====
// Validate and decode a GPT Table descriptor

(GPCF, GPTTable) DecodeGPTTable(PGSe pgs, bits(64) gpt_entry)
assert gpt_entry<3:0> == GPT\_Table;
GPTTable result;

if !IsZero(gpt_entry<63:52,11:4>) then
    return (GPCF\_Walk, GPTTable UNKNOWN);

l0sz = GPTL0Size();
integer p;
case pgs of
    when PGS\_4KB p = 12;
    when PGS\_16KB p = 14;
    when PGS\_64KB p = 16;
    otherwise Unreachable();

if !IsZero(gpt_entry<(l0sz-p)-2:12>) then
    return (GPCF\_Walk, GPTTable UNKNOWN);

case pgs of
    when PGS\_4KB result.address = gpt_entry<55:17>:Zeros(17);
    when PGS\_16KB result.address = gpt_entry<55:15>:Zeros(15);
    when PGS\_64KB result.address = gpt_entry<55:13>:Zeros(13);
    otherwise Unreachable();

// The address must be within the range covered by the GPT
if AbovePPS(result.address) then
    return (GPCF\_AddressSize, GPTTable UNKNOWN);

return (GPCF\_None, result);
```

### Library pseudocode for shared/translation/gpc/DecodePGS

```
// DecodePGS()
// =====

PGSe DecodePGS(bits(2) pgs)
  case pgs of
    when '00' return PGS\_4KB;
    when '10' return PGS\_16KB;
    when '01' return PGS\_64KB;
    otherwise Unreachable();
```

### Library pseudocode for shared/translation/gpc/DecodePPS

```
// DecodePPS()
// =====
// Size of region protected by the GPT, in bits.

integer DecodePPS()
  case GPCCR_EL3.PPS of
    when '000' return 32;
    when '001' return 36;
    when '010' return 40;
    when '011' return 42;
    when '100' return 44;
    when '101' return 48;
    when '110' return 52;
    otherwise Unreachable();
```

### Library pseudocode for shared/translation/gpc/GPCFault

```
// GPCFault()
// =====
// Constructs and returns a GPCF

GPCFRecord GPCFault(GPCF gpf, integer level)
  GPCFRecord fault;
  fault.gpf = gpf;
  fault.level = level;
  return fault;
```

### Library pseudocode for shared/translation/gpc/GPCNoFault

```
// GPCNoFault()
// =====
// Returns the default properties of a GPCF that does not represent a fault

GPCFRecord GPCNoFault()
  GPCFRecord result;
  result.gpf = GPCF\_None;
  return result;
```

## Library pseudocode for shared/translation/gpc/GPCRegistersConsistent

```
// GPCRegistersConsistent()
// =====
// Returns whether the GPT registers are configured correctly.
// This returns false if any fields select a Reserved value.

boolean GPCRegistersConsistent()

    // Check for Reserved register values
    if GPCCR_EL3.PPS == '111' || DecodePPS\(\) > AArch64.PAMax\(\) then
        return FALSE;
    if GPCCR_EL3.PGS == '11' then
        return FALSE;
    if GPCCR_EL3.SH == '01' then
        return FALSE;

    // Inner and Outer Non-cacheable requires Outer Shareable
    if GPCCR_EL3.<ORGN, IRGN> == '0000' && GPCCR_EL3.SH != '10' then
        return FALSE;

    return TRUE;
```

## Library pseudocode for shared/translation/gpc/GPICheck

```
// GPICheck()
// =====
// Returns whether an access to a given physical address space is permitted
// given the configured GPI value.
// paspace: Physical address space of the access
// gpi: Value read from GPT for the access

boolean GPICheck(PASpace paspace, bits(4) gpi)

    case gpi of
        when GPT\_NoAccess    return FALSE;
        when GPT\_Secure     assert HaveSecureEL2Ext\(\); return paspace == PAS\_Secure;
        when GPT\_NonSecure   return paspace == PAS\_NonSecure;
        when GPT\_Root       return paspace == PAS\_Root;
        when GPT\_Realm      return paspace == PAS\_Realm;
        when GPT\_Any        return TRUE;
        otherwise            Unreachable\(\);
```

## Library pseudocode for shared/translation/gpc/GPIIndex

```
// GPIIndex()
// =====

integer GPIIndex(bits(56) pa)
    case DecodePGS(GPCCR_EL3.PGS) of
        when PGS\_4KB    return UInt(pa<15:12>);
        when PGS\_16KB   return UInt(pa<17:14>);
        when PGS\_64KB   return UInt(pa<19:16>);
        otherwise       Unreachable\(\);
```



### Library pseudocode for shared/translation/gpc/GPIValid

```
// GPIValid()
// =====
// Returns whether a given value is a valid encoding for a GPI value

boolean GPIValid(bits(4) gpi)
  if gpi == GPT\_Secure then
    return HaveSecureEL2Ext();

  return gpi IN {GPT\_NoAccess,
                 GPT\_NonSecure,
                 GPT\_Root,
                 GPT\_Realm,
                 GPT\_Any};
```

### Library pseudocode for shared/translation/gpc/GPTL0Size

```
// GPTL0Size()
// =====
// Returns number of bits covered by a level 0 GPT entry

integer GPTL0Size()
  case GPCCR_EL3.LOGPTSZ of
    when '0000' return GPTRange\_1GB;
    when '0100' return GPTRange\_16GB;
    when '0110' return GPTRange\_64GB;
    when '1001' return GPTRange\_512GB;
    otherwise Unreachable();
  return 30;
```

### Library pseudocode for shared/translation/gpc/GPTLevel0Index

```
// GPTLevel0Index()
// =====
// Compute the level 0 index based on input PA.

integer GPTLevel0Index(bits(56) pa)
  // Input address and index bounds
  pps = DecodePPS();
  l0sz = GPTL0Size();
  if pps <= l0sz then
    return 0;

  return UInt(pa<pps-1:l0sz>);
```

### Library pseudocode for shared/translation/gpc/GPTLevel1Index

```
// GPTLevel1Index()
// =====
// Compute the level 1 index based on input PA.

integer GPTLevel1Index(bits(56) pa)
  // Input address and index bounds
  l0sz = GPTL0Size();
  case DecodePGS(GPCCR_EL3.PGS) of
    when PGS\_4KB return UInt(pa<l0sz-1:16>);
    when PGS\_16KB return UInt(pa<l0sz-1:18>);
    when PGS\_64KB return UInt(pa<l0sz-1:20>);
    otherwise Unreachable();
```



```

// GPTWalk()
// =====
// Get the GPT entry for a given physical address, pa

(GPCFRecord, GPTEntry) GPTWalk(bits(56) pa, AccessDescriptor accdesc)

    // GPT base address
    bits(56) base;
    pgs = DecodePGS(GPCCR_EL3.PGS);

    // The level 0 GPT base address is aligned to the greater of:
    // * the size of the level 0 GPT, determined by GPCCR_EL3.{PPS, LOGPTSZ}.
    // * 4KB
    base = ZeroExtend(GPTBR_EL3.BADDR:Zeros(12), 56);
    pps = DecodePPS();
    l0sz = GPTL0Size();
    integer alignment = Max((pps - l0sz) + 3, 12);
    base = base AND NOT ZeroExtend(Ones(alignment), 56);

    AccessDescriptor gptaccdesc = CreateAccDescGPTW(accdesc);

    // Access attributes and address for GPT fetches
    AddressDescriptor gptaddrdesc;
    gptaddrdesc.memattrs = WalkMemAttrs(GPCCR_EL3.SH, GPCCR_EL3.ORG, GPCCR_EL3.IRG);
    gptaddrdesc.fault = NoFault(gptaccdesc);

    // Address of level 0 GPT entry
    gptaddrdesc.paddress.paspace = PAS\_Root;
    gptaddrdesc.paddress.address = base + GPTLevel0Index(pa) * 8;

    // Fetch LOGPT entry
    bits(64) level_0_entry;
    PhysMemRetStatus memstatus;
    (memstatus, level_0_entry) = PhysMemRead(gptaddrdesc, 8, gptaccdesc);
    if IsFault(memstatus) then
        return (GPCFault(GPCF\_EABT, 0), GPTEntry UNKNOWN);

    GPTEntry result;
    GPTTable table;
    GPCF gpf;
    case level_0_entry<3:0> of
        when GPT\_Block
            // Decode the GPI value and return that
            (gpf, result) = DecodeGPTBlock(pgs, level_0_entry);
            result.pa = pa;
            return (GPCFault(gpf, 0), result);
        when GPT\_Table
            // Decode the table entry and continue walking
            (gpf, table) = DecodeGPTTable(pgs, level_0_entry);
            if gpf != GPCF\_None then
                return (GPCFault(gpf, 0), GPTEntry UNKNOWN);
            otherwise
                // GPF - invalid encoding
                return (GPCFault(GPCF\_Walk, 0), GPTEntry UNKNOWN);

    // Must be a GPT Table entry
    assert level_0_entry<3:0> == GPT\_Table;

    // Address of level 1 GPT entry
    offset = GPTLevel1Index(pa) * 8;
    gptaddrdesc.paddress.address = table.address + offset;

    // Fetch L1GPT entry
    bits(64) level_1_entry;
    (memstatus, level_1_entry) = PhysMemRead(gptaddrdesc, 8, gptaccdesc);
    if IsFault(memstatus) then
        return (GPCFault(GPCF\_EABT, 1), GPTEntry UNKNOWN);

    case level_1_entry<3:0> of
        when GPT\_Contig

```

```

        (gpf, result) = DecodeGPTContiguous(pgs, level_1_entry);
    otherwise
        gpi_index = GPIIndex(pa);
        (gpf, result) = DecodeGPTGranules(pgs, gpi_index, level_1_entry);

    if gpf != GPCF\_None then
        return (GPCFault(gpf, 1), GPTEntry UNKNOWN);

    result.pa = pa;
    return (GPCNoFault(), result);

```

## Library pseudocode for shared/translation/gpc/GranuleProtectionCheck

```

// GranuleProtectionCheck()
// =====
// Returns whether a given access is permitted, according to the
// granule protection check.
// addrdesc and accdesc describe the access to be checked.

GPCFRecord GranuleProtectionCheck(AddressDescriptor addrdesc, AccessDescriptor accdesc)

    assert HaveRME();
    // The address to be checked
    address = addrdesc.paddress;

    // Bypass mode - all accesses pass
    if GPCCR_EL3.GPC == '0' then
        return GPCNoFault();

    // Configuration consistency check
    if !GPCRegistersConsistent() then
        return GPCFault(GPCF\_Walk, 0);

    // Input address size check
    if AbovePPS(address.address) then
        if address.paspace == PAS\_NonSecure then
            return GPCNoFault();
        else
            return GPCFault(GPCF\_Fail, 0);

    // GPT base address size check
    bits(56) gpt_base = ZeroExtend(GPTBR_EL3.BADDR:Zeros(12), 56);
    if AbovePPS(gpt_base) then
        return GPCFault(GPCF\_AddressSize, 0);

    // GPT lookup
    (gpcf, gpt_entry) = GPTWalk(address.address, accdesc);
    if gpcf.gpf != GPCF\_None then
        return gpcf;

    // Check input physical address space against GPI
    permitted = GPICheck(address.paspace, gpt_entry.gpi);

    if !permitted then
        gpcf = GPCFault(GPCF\_Fail, gpt_entry.level);
        return gpcf;

    // Check passed

    return GPCNoFault();

```

## Library pseudocode for shared/translation/gpc/PGS

```
// PGS
// ===
// Physical granule size

enumeration PGSe {
    PGS_4KB,
    PGS_16KB,
    PGS_64KB
};
```

## Library pseudocode for shared/translation/gpc/Table

```
constant bits(4) GPT_NoAccess  = '0000';
constant bits(4) GPT_Table     = '0011';
constant bits(4) GPT_Block     = '0001';
constant bits(4) GPT_Contig    = '0001';
constant bits(4) GPT_Secure    = '1000';
constant bits(4) GPT_NonSecure = '1001';
constant bits(4) GPT_Root      = '1010';
constant bits(4) GPT_Realm     = '1011';
constant bits(4) GPT_Any       = '1111';
constant integer GPTRange_4KB   = 12;
constant integer GPTRange_16KB  = 14;
constant integer GPTRange_64KB  = 16;
constant integer GPTRange_2MB   = 21;
constant integer GPTRange_32MB  = 25;
constant integer GPTRange_512MB = 29;
constant integer GPTRange_1GB   = 30;
constant integer GPTRange_16GB  = 34;
constant integer GPTRange_64GB  = 36;
constant integer GPTRange_512GB = 39;

type GPTTable is (
    bits(56) address      // Base address of next table
)

type GPTEntry is (
    bits(4)  gpi,          // GPI value for this region
    integer  size,         // Region size
    integer  contig_size,  // Contiguous region size
    integer  level,        // Level of GPT lookup
    bits(56) pa            // PA uniquely identifying the GPT entry
)
```

## Library pseudocode for shared/translation/translation/S1TranslationRegime

```
// S1TranslationRegime()
// =====
// Stage 1 translation regime for the given Exception level

bits(2) S1TranslationRegime(bits(2) el)
    if el != EL0 then
        return el;
    elsif HaveEL\(EL3\) && ELUsingAArch32\(EL3\) && SCR.NS == '0' then
        return EL3;
    elsif HaveVirtHostExt\(\) && ELIsInHost\(el\) then
        return EL2;
    else
        return EL1;

// S1TranslationRegime()
// =====
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the System register accessors (SCTLR_ELx[], etc.) implicitly
// return the correct value.

bits(2) S1TranslationRegime()
    return S1TranslationRegime(PSTATE.EL);
```

## Library pseudocode for shared/translation/vmsa/AddressDescriptor

```
constant integer FINAL_LEVEL = 3;

// AddressDescriptor
// =====
// Descriptor used to access the underlying memory array.

type AddressDescriptor is (
    FaultRecord      fault,          // fault.statuscode indicates whether the address is valid
    MemoryAttributes memattrs,
    FullAddress      paddress,
    boolean          slassured,     // Stage 1 Assured Translation Property
    boolean          s2fslmro,     // Stage 2 MRO permission for Satge 1
    bits(16)         mecid,        // FEAT_MEC: Memory Encryption Context ID
    bits(64)         vaddress
)
```

## Library pseudocode for shared/translation/vmsa/ContiguousSize

```
// ContiguousSize()
// =====
// Return the number of entries log 2 marking a contiguous output range

integer ContiguousSize(bit d128, TGx tgx, integer level)
    if d128 == '1' then
        case tgx of
            when TGx\_4KB
                assert level IN {1, 2, 3};
                return if level == 1 then 2 else 4;
            when TGx\_16KB
                assert level IN {1, 2, 3};
                if level == 1 then
                    return 2;
                elsif level == 2 then
                    return 4;
                else
                    return 6;
            when TGx\_64KB
                assert level IN {2, 3};
                return if level == 2 then 6 else 4;
        else
            case tgx of
                when TGx\_4KB
                    assert level IN {1, 2, 3};
                    return 4;
                when TGx\_16KB
                    assert level IN {2, 3};
                    return if level == 2 then 5 else 7;
                when TGx\_64KB
                    assert level IN {2, 3};
                    return 5;
```

## Library pseudocode for shared/translation/vmsa/CreateAddressDescriptor

```
// CreateAddressDescriptor()
// =====
// Set internal members for address descriptor type to valid values

AddressDescriptor CreateAddressDescriptor(bits(64) va, FullAddress pa,
                                          MemoryAttributes memattrs)

    AddressDescriptor addrdesc;

    addrdesc.paddress = pa;
    addrdesc.vaddress = va;
    addrdesc.memattrs = memattrs;
    addrdesc.fault    = NoFault();
    addrdesc.slassured = FALSE;

    return addrdesc;
```

## Library pseudocode for shared/translation/vmsa/CreateFaultyAddressDescriptor

```
// CreateFaultyAddressDescriptor()
// =====
// Set internal members for address descriptor type with values indicating error

AddressDescriptor CreateFaultyAddressDescriptor(bits(64) va, FaultRecord fault)
    AddressDescriptor addrdesc;

    addrdesc.vaddress = va;
    addrdesc.fault    = fault;

    return addrdesc;
```

## Library pseudocode for shared/translation/vmsa/DecodePASpace

```
// DecodePASpace()
// =====
// Decode the target PA Space

PASpace DecodePASpace (bit nse, bit ns)
    case nse:ns of
        when '00'    return PAS\_Secure;
        when '01'    return PAS\_NonSecure;
        when '10'    return PAS\_Root;
        when '11'    return PAS\_Realm;
```

## Library pseudocode for shared/translation/vmsa/DescriptorType

```
// DescriptorType
// =====
// Translation table descriptor formats

enumeration DescriptorType {
    DescriptorType_Table,
    DescriptorType_Leaf,
    DescriptorType_Invalid
};
```

## Library pseudocode for shared/translation/vmsa/Domains

```
constant bits(2) Domain_NoAccess = '00';
constant bits(2) Domain_Client   = '01';
constant bits(2) Domain_Manager  = '11';
```

## Library pseudocode for shared/translation/vmsa/FetchDescriptor

```
// FetchDescriptor()
// =====
// Fetch a translation table descriptor

(FaultRecord, bits(N)) FetchDescriptor(bit ee, AddressDescriptor walkaddress,
                                       AccessDescriptor walkaccess, FaultRecord fault_in,
                                       integer N)

    // 32-bit descriptors for AArch32 Short-descriptor format
    // 64-bit descriptors for AArch64 or AArch32 Long-descriptor format
    // 128-bit descriptors for AArch64 when FEAT_D128 is set and {V}TCR_ELx.d128 is set
    assert N == 32 || N == 64 || N == 128;
    bits(N) descriptor;
    FaultRecord fault = fault_in;

    if HaveRME() then
        fault.gpcf = GranuleProtectionCheck(walkaddress, walkaccess);
        if fault.gpcf.gpf != GPCF\_None then
            fault.statuscode = Fault\_GPCFOnWalk;
            fault.paddress   = walkaddress.paddress;
            fault.gpcfs2walk = fault.secondstage;
            return (fault, bits(N) UNKNOWN);

    PhysMemRetStatus memstatus;
    (memstatus, descriptor) = PhysMemRead(walkaddress, N DIV 8, walkaccess);
    if IsFault(memstatus) then
        boolean iswrite = FALSE;
        fault = HandleExternalTTWAbort(memstatus, iswrite, walkaddress,
                                       walkaccess, N DIV 8, fault);

        if IsFault(fault.statuscode) then
            return (fault, bits(N) UNKNOWN);

    if ee == '1' then
        descriptor = BigEndianReverse(descriptor);

    return (fault, descriptor);
```



## Library pseudocode for shared/translation/vmsa/HasUnprivileged

```
// HasUnprivileged()
// =====
// Returns whether a translation regime serves EL0 as well as a higher EL

boolean HasUnprivileged(Regime regime)
    return (regime IN {
        Regime\_EL20,
        Regime\_EL30,
        Regime\_EL10
    });
```

## Library pseudocode for shared/translation/vmsa/Regime

```
// Regime
// =====
// Translation regimes

enumeration Regime {
    Regime_EL3,           // EL3
    Regime_EL30,          // EL3&0 (PL1&0 when EL3 is AArch32)
    Regime_EL2,           // EL2
    Regime_EL20,          // EL2&0
    Regime_EL10           // EL1&0
};
```

## Library pseudocode for shared/translation/vmsa/RegimeUsingAArch32

```
// RegimeUsingAArch32()
// =====
// Determine if the EL controlling the regime executes in AArch32 state

boolean RegimeUsingAArch32(Regime regime)
    case regime of
        when Regime\_EL10 return ELUsingAArch32(EL1);
        when Regime\_EL30 return TRUE;
        when Regime\_EL20 return FALSE;
        when Regime\_EL2 return ELUsingAArch32(EL2);
        when Regime\_EL3 return FALSE;
```

## Library pseudocode for shared/translation/vmsa/S1TTWParams

```
// S1TTWParams
// =====
// Register fields corresponding to stage 1 translation
// For A32-VMSA, if noted, they correspond to A32-LPAE (Long descriptor format)

type S1TTWParams is (
// A64-VMSA exclusive parameters
    bit          ha,          // TCR_ELx.HA
    bit          hd,          // TCR_ELx.HD
    bit          tbi,         // TCR_ELx.TBI{x}
    bit          tbid,        // TCR_ELx.TBID{x}
    bit          nfd,         // TCR_EL1.NFDx or TCR_EL2.NFDx when HCR_EL2.E2H == '1'
    bit          e0pd,        // TCR_EL1.E0PDx or TCR_EL2.E0PDx when HCR_EL2.E2H == '1'
    bit          dl28,        // TCR_ELx.D128
    bit          aie,         // (TCR2_ELx/TCR_EL3).AIE
    MAIRType     mair2,       // MAIR2_ELx
    bit          ds,          // TCR_ELx.DS
    bits(3)      ps,          // TCR_ELx.{I}PS
    bits(6)      txsz,        // TCR_ELx.TxSZ
    bit          epan,        // SCTLR_EL1.EPAN or SCTLR_EL2.EPAN when HCR_EL2.E2H == '1'
    bit          dct,         // HCR_EL2.DCT
    bit          nv1,         // HCR_EL2.NV1
    bit          cmow,        // SCTLR_EL1.CMOW or SCTLR_EL2.CMOW when HCR_EL2.E2H == '1'
    bit          pnch,        // TCR{2}_ELx.PnCH
    bit          disch,       // TCR{2}_ELx.DisCH
    bit          haft,        // TCR{2}_ELx.HAFT
    bit          mtX,         // TCR_ELx.MTX{y}
    bits(2)      skl,         // TCR_ELx.SKL
    bit          pie,         // TCR2_ELx.PIE or TCR_EL3.PIE
    S1PIRType    pir,         // PIR_ELx
    S1PIRType    pire0,       // PIRE0_EL1 or PIRE0_EL2 when HCR_EL2.E2H == '1'
    bit          emec,        // SCTLR2_EL2.EMEC or SCTLR2_EL3.EMEC
    bit          amec,        // TCR2_EL2.AMEC0 or TCR2_EL2.AMEC1 when HCR_EL2.E2H == '1'

// A32-VMSA exclusive parameters
    bits(3)      t0sz,        // TTBCR.T0SZ
    bits(3)      t1sz,        // TTBCR.T1SZ
    bit          uwxn,        // SCTLR.UWXN

// Parameters common to both A64-VMSA & A32-VMSA (A64/A32)
    TGx          tgX,          // TCR_ELx.TGx      / Always TGx_4KB
    bits(2)      irgn,        // TCR_ELx.IRGNx   / TTBCR.IRGNx or HTCR.IRGN0
    bits(2)      orgn,        // TCR_ELx.ORGNx   / TTBCR.ORGnx or HTCR.ORGNO
    bits(2)      sh,          // TCR_ELx.SHx     / TTBCR.SHx or HTCR.SH0
    bit          hpd,         // TCR_ELx.HPD{x}  / TTBCR2.HPDx or HTCR.HPD
    bit          ee,          // SCTLR_ELx.EE    / SCTLR.EE or HSCTLR.EE
    bit          wxn,         // SCTLR_ELx.WXN   / SCTLR.WXN or HSCTLR.WXN
    bit          ntlsmd,      // SCTLR_ELx.nTLSMD / SCTLR.nTLSMD or HSCTLR.nTLSMD
    bit          dc,          // HCR_EL2.DC      / HCR.DC
    bit          sif,         // SCR_EL3.SIF     / SCR.SIF
    MAIRType     mair         // MAIR_ELx        / MAIR1:MAIR0 or HMAIR1:HMAIR0
)
```

## Library pseudocode for shared/translation/vmsa/S2TTWParams

```
// S2TTWParams
// =====
// Register fields corresponding to stage 2 translation.

type S2TTWParams is (
// A64-VMSA exclusive parameters
    bit      ha,          // VTCR_EL2.HA
    bit      hd,          // VTCR_EL2.HD
    bit      sl2,         // V{S}TCR_EL2.SL2
    bit      ds,          // VTCR_EL2.DS
    bit      dl28,        // VTCR_EL2.D128
    bit      sw,          // VSTCR_EL2.SW
    bit      nsw,         // VTCR_EL2.NSW
    bit      sa,          // VSTCR_EL2.SA
    bit      nsa,         // VTCR_EL2.NSA
    bits(3)   ps,         // VTCR_EL2.PS
    bits(6)   txsz,       // V{S}TCR_EL2.T0SZ
    bit      fw,         // HCR_EL2.PTW
    bit      cmow,        // HCRX_EL2.CMOW
    bits(2)   skl,        // VTCR_EL2.SKL
    bit      s2pie,       // VTCR_EL2.S2PIE
    S2PIRType s2pir,     // S2PIR_EL2
    bit      tl0,         // VTCR_EL2.TL0
    bit      tl1,         // VTCR_EL2.TL1
    bit      assuredonly, // VTCR_EL2.AssuredOnly
    bit      haft,        // VTCR_EL2.HAFT
    bit      emec,        // SCTLR2_EL2.EMEC

// A32-VMSA exclusive parameters
    bit      s,          // VTCR.S
    bits(4)   t0sz,       // VTCR.T0SZ

// Parameters common to both A64-VMSA & A32-VMSA if implemented (A64/A32)
    TGx      tgx,         // V{S}TCR_EL2.TG0 / Always TGx_4KB
    bits(2)   sl0,        // V{S}TCR_EL2.SL0 / VTCR.SL0
    bits(2)   irgn,       // VTCR_EL2.IRGN0 / VTCR.IRGN0
    bits(2)   orgn,       // VTCR_EL2.ORGNO / VTCR.ORGNO
    bits(2)   sh,         // VTCR_EL2.SH0 / VTCR.SH0
    bit      ee,          // SCTLR_EL2.EE / HSCTLR.EE
    bit      ptw,         // HCR_EL2.PTW / HCR.PTW
    bit      vm,          // HCR_EL2.VM / HCR.VM
)
```

## Library pseudocode for shared/translation/vmsa/SDFType

```
// SDFType
// =====
// Short-descriptor format type

enumeration SDFType {
    SDFType_Table,
    SDFType_Invalid,
    SDFType_Supersection,
    SDFType_Section,
    SDFType_LargePage,
    SDFType_SmallPage
};
```

## Library pseudocode for shared/translation/vmsa/SecurityStateForRegime

```
// SecurityStateForRegime()
// =====
// Return the Security State of the given translation regime

SecurityState SecurityStateForRegime(Regime regime)
    case regime of
        when Regime\_EL3      return SecurityStateAtEL(EL3);
        when Regime\_EL30     return SS\_Secure; // A32 EL3 is always Secure
        when Regime\_EL2      return SecurityStateAtEL(EL2);
        when Regime\_EL20     return SecurityStateAtEL(EL2);
        when Regime\_EL10     return SecurityStateAtEL(EL1);
```

## Library pseudocode for shared/translation/vmsa/StageOA

```
// StageOA()
// =====
// Given the final walk state (a page or block descriptor), map the untranslated
// input address bits to the output address

FullAddress StageOA(bits(64) ia, bit d128, TGx tgx, TTWState walkstate)
    // Output Address
    FullAddress oa;
    integer csize;

    tsize = TranslationSize(d128, tgx, walkstate.level);
    if walkstate.contiguous == '1' then
        csize = ContiguousSize(d128, tgx, walkstate.level);
    else
        csize = 0;

    ia_msb = tsize + csize;
    oa.paspace = walkstate.baseaddress.paspace;
    oa.address = walkstate.baseaddress.address<55:ia_msb>:ia<ia_msb-1:0>;

    return oa;
```

## Library pseudocode for shared/translation/vmsa/TGx

```
// TGx
// ===
// Translation granules sizes

enumeration TGx {
    TGx_4KB,
    TGx_16KB,
    TGx_64KB
};
```

## Library pseudocode for shared/translation/vmsa/TGxGranuleBits

```
// TGxGranuleBits()
// =====
// Retrieve the address size, in bits, of a granule

integer TGxGranuleBits(TGx tgx)
    case tgx of
        when TGx\_4KB  return 12;
        when TGx\_16KB return 14;
        when TGx\_64KB return 16;
```

## Library pseudocode for shared/translation/vmsa/TLBContext

```
// TLBContext
// =====
// Translation context compared on TLB lookups and invalidations, promoting a TLB hit on match

type TLBContext is (
    SecurityState ss,
    Regime regime,
    bits(16) vmid,
    bits(16) asid,
    bit nG,
    PASpace ipaspace, // Used in stage 2 lookups & invalidations only
    boolean includes_s1,
    boolean includes_s2,
    boolean includes_gpt,
    bits(64) ia, // Input Address
    TGx tg,
    bit cnp,
    integer level, // Assist TLBI level hints (FEAT_TTL)
    boolean isd128,
    bit xs // XS attribute (FEAT_XS)
)
```

## Library pseudocode for shared/translation/vmsa/TLBRecord

```
// TLBRecord
// =====
// Translation output as a TLB payload

type TLBRecord is (
    TLBContext context,
    TTWState walkstate,
    integer blocksize, // Number of bits directly mapped from IA to OA
    integer contigsize, // Number of entries log 2 marking a contiguous output range
    bits(128) s1descriptor, // Stage 1 leaf descriptor in memory (valid if the TLB caches stage 1)
    bits(128) s2descriptor // Stage 2 leaf descriptor in memory (valid if the TLB caches stage 2)
)
```

## Library pseudocode for shared/translation/vmsa/TTWState

```
// TTWState
// =====
// Translation table walk state

type TTWState is (
    boolean istable,
    integer level,
    FullAddress baseaddress,
    bit contiguous,
    boolean s1assured, // Stage 1 Assured Translation Property
    bit s2assuredonly, // Stage 2 AssuredOnly attribute
    bit disch, // Stage 1 Disable Contiguous Hint
    bit nG,
    bit guardedpage,
    SDFTType sdftype, // AArch32 Short-descriptor format walk only
    bits(4) domain, // AArch32 Short-descriptor format walk only
    MemoryAttributes memattrs,
    Permissions permissions
)
```

## Library pseudocode for shared/translation/vmsa/TranslationRegime

```
// TranslationRegime()
// =====
// Select the translation regime given the target EL and PE state

Regime TranslationRegime(bits(2) el)
    if el == EL3 then
        return if ELUsingAArch32(EL3) then Regime\_EL30 else Regime\_EL3;
    elsif el == EL2 then
        return if ELIsInHost(EL2) then Regime\_EL20 else Regime\_EL2;
    elsif el == EL1 then
        return Regime\_EL10;
    elsif el == EL0 then
        if CurrentSecurityState() == SS\_Secure && ELUsingAArch32(EL3) then
            return Regime\_EL30;
        elsif ELIsInHost(EL0) then
            return Regime\_EL20;
        else
            return Regime\_EL10;
    else
        Unreachable();
```

## Library pseudocode for shared/translation/vmsa/TranslationSize

```
// TranslationSize()
// =====
// Compute the number of bits directly mapped from the input address
// to the output address

integer TranslationSize(bit d128, TGx tgx, integer level)
    granulebits = TGxGranuleBits(tgx);
    descsize2 = if d128 == '1' then 4 else 3;
    blockbits = (FINAL\_LEVEL - level) * (granulebits - descsize2);

    return granulebits + blockbits;
```

## Library pseudocode for shared/translation/vmsa/UseASID

```
// UseASID()
// =====
// Determine whether the translation context for the access requires ASID or is a global entry

boolean UseASID(TLBContext accesscontext)
    return HasUnprivileged(accesscontext.regime);
```

## Library pseudocode for shared/translation/vmsa/UseVMID

```
// UseVMID()
// =====
// Determine whether the translation context for the access requires VMID to match a TLB entry

boolean UseVMID(TLBContext accesscontext)
    return accesscontext.regime == Regime\_EL10 && EL2Enabled();
```

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode v2023-06\_rel, sve v2023-06\_rel ; Build timestamp: 2023-07-04T19:42

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.